MATHEMATICS WRITTEN IN SAND — the hp-15C, Intel 8087, etc.

W. Kahan, University of California @ Berkeley

ABSTRACT: Simplicity is a Virtue; yet we continue to cram ever more complicated circuits ever more densely into silicon chips, hoping all the while that their internal complexity will promote simplicity of use. This paper exhibits how well that hope has been fulfilled by several inexpensive devices widely used nowadays for numerical computation. One of them is the Hewlett-Packard hp-15C programmable shirt-pocket calculator, on which only a few keys need be pressed to perform tasks like these:

Real and Complex arithmetic, including the elementary transcendental functions and their inverses; Matrix arithmetic including inverse, transpose, determinant, residual, norms, prompted input/output and complex-real conversion; Solve an equation and evaluate an Integral numerically; simple statistics; Γ and combinatorial functions; ...

For instance, a stroke of its [1/X] key inverts an 8x8 matrix of 10-sig.-dec. numbers in 90 sec. This calculator costs under $100 by mail-order.

Mathematically dense circuitry is also found in Intel's 8087 coprocessor chip, currently priced below $200, which has for two years augmented the instruction repertoire of the 8086 and 8088 microcomputer chips to cope with ...

Three binary floating-point formats 32, 64 and 80 bits wide; three binary integer formats 16, 32 and 64 bits wide; 18-digit BCDecimal integers; rational arithmetic, square root, format conversion and exception handling all in conformity with p754, the proposed IEEE arithmetic standard (see "Computer" Mar. 1, 1981); the kernels of transcendental functions exp, log, tan and arctan; and an internal stack of eight registers each 80 bits wide.

For instance, the 8087 has been used to invert a 100x100 matrix of 64-bit floating-point numbers in 90 sec. Among the machines that can use this chip are the widely distributed IBM Personal Computers, each containing a socket already wired for an 8087. Several other manufacturers now produce arithmetic engines that, like the 8087, conform to the proposed IEEE arithmetic standard, so software that exploits its refined arithmetic properties should be widespread soon.

As sophisticated mathematical operations come into use ever more widely, mathematical proficiency appears to rise; in a sense it actually declines. Computations formerly reserved for experts lie now within reach of whoever might benefit from them regardless of how little mathematics he understands; and that little is more likely to have been gleaned from handbooks for calculators and personal computers than from professors. This trend is pronounced among users of financial calculators like the hp-12C. Such trends ought to affect what and how we teach, as well as how we use mathematics, regardless of whether large fast computers, hitherto dedicated mostly to speed, ever catch up with some smaller machines' progress towards mathematical robustness and convenience.

Prepared for the Joint Statistical Meetings of the ASA-ENAR-WNAR-IMS-SSC held in Toronto, Canada, August 15-18, 1983.

INTRODUCTION: As a schoolboy in Toronto I was taught to cherish each advance in Science in so far as it enabled us to know more while obliging us to memorize less. By that criterion, albeit oversimplified, the technological advances that now rain computer hardware and software upon us do not yet constitute an advance in Science, not so long as they are accompanied by a hail of needless inconsistencies and incompatibilities. Hardest to explain, in devices presumably dedicated to numerical computation, are the arithmetical anomalies that arise from defective mathematical doctrines rather than from mere oversights. For instance, the following table was printed out by VisiCorp's spread-sheet program called "VisiCalc 1.10" run on an IBM Personal Computer :

A	B = A/3	C = 3*B	A - C	A/2 - C + A/2
100	33.3333333333	99.9999999999	.00000001	.0000000001
1000	333.33333333	999.99999999	.00000001	.00000001
10000	3333.33333333	9999.99999999	.000001	.00000001
100000	33333.333333	99999.999999	.000001	.000001
1000000	333333.333333	999999.999999	.0001	.000001
10000000	3333333.3333	9999999.9999	.0001	.0001
100000000	33333333.3333	99999999.9999	.01	.0001
1000000000	333333333.33	999999999.99	.01	.01

Perhaps roundoff could account plausibly for the second column's jaggedness; but how can errors in the fourth column be reconciled with correct values in the fifth? Imagine explaining them to a Computer Science class in programming:

"To calculate (A - C) much more accurately, evaluate (A/2 - C + A/2) instead because ..."

Since a far-fetched explanation is undignified, one might prefer to believe these anomalies are inconsequential and need no explanation. That belief induced some anonymous programmer to deem them acceptable as a side-effect of a shortened and faster program that performs arithmetic for VisiCalc in radix 100 instead of 10 and drops a digit prematurely. Actually, the program is only imperceptibly shorter and faster, but its anomalies are manifest and, as examples below will show, malignant. Fortunately, a wide range of calculators and computers, especially those that conform to the IEEE's proposed standards p754 and p854 for floating-point arithmetic, do not suffer from paradoxical roundoff like that displayed above. Those machines and standards are part of what this paper is about.

Anomalies generally undermine economical thought, thereby undermining the integrity of software and inflating its cost. The worst anomalies can be kept out of computers. When they do intrude they are not always accidental; too often they follow from design decisions induced by misconceptions widely taught as rules of thumb about what to neglect in approximate computation. Refutations of those misconceptions abound in the literature [1,2,3,4,5,6] but cannot help someone who has

Reprinted from the 1983 Statistical Computing Section Proceedings of the American Statistical Association

not read them, who believes every elementary subject must be obvious, and whose mathematical experience is too narrow to support sound judgments. Here is another domain where our failure to teach mathematics effectively to a past generation comes home to roost.

I do not allege that mathematical education has failed entirely. For most, education succeeds as soon as they can follow a formula chosen for them by Experience or Authority. A few, captivated by the beauty or abstractness of the subject, espouse mathematics to escape the mundane, and then need little help from the likes of me. But many who endure two years of College Mathematics do so in the hope that it will help them explore and conquer other domains. They would crown Mathematics "Queen of the Sciences" more for her power to illuminate her applications than for her beauty or abstractness. Alas, they lack the mathematical experience out of which grow first the abstractions and then the conviction that these are the source of illumination. Lacking too is time we can spend together exploring examples instead of exchanging mere formalities. So, when I try in class to illuminate for them the power and the beauty of the subject I love, abstractions that sum up lifetimes of experience turn to chalk dust faster than my students can copy, much less learn. What will defend them against me and my kind?

Rather than have to copy the received word, students are entitled to experiment with mathematical phenomena, discover more of them, and then read how our predecessors discovered even more. Students need inexpensive apparatus analogous to the instruments and glassware in Physics and Chemistry laboratories, but designed to combat the drudgery that inhibits exploration. This role is the first that I envisaged for the hp-15C shirt-pocket calculator when it was being designed. Later, among students who find it helpful for their Engineering and Science assignments, I hoped a few might wonder how it works and why; some of these would become computer scientists and applied mathematicians all the more comfortable with important ideas and techniques for having encountered them in their own calculators. Those ideas are part of what this paper is about

This paper does not say just that computers are smaller, cheaper, faster and more capacious. It tells how some machines convey mathematical ideas to a far wider audience than used to benefit from them. What Archimedes wrote in sand* could be read by only a few before it blew away. Written on paper, his ideas have been read by myriads and will be read by myriads more. When written into silicon chips, his ideas and their cousins serve the needs of hundreds of thousands now, and soon millions. (* Sand is mostly Silicon Dioxide.)

WHO'S TO BLAME? Conventional wisdom says that in those rare and pathological instances when computed results are found to be wrong because of roundoff, the right results can always be gotten by recomputation, either carrying more figures in what is otherwise the same procedure as before, or via a different and more "stable" numerical algorithm that could be very hard to find. This conventional wisdom begs three questions:

How can anybody tell when and why results are wrong?
Who is responsible for finding and correcting wrong results?
Will carrying more figures *always* attenuate roundoff?

The same imperatives that move us to share scientific knowledge force us to share computer software. When we share knowledge we share an understanding that leaves intact each individual's responsibility for the consequences of the use of that knowledge. But when we share software, responsibility diffuses; were you obliged to understand in detail the program you got from me, you might as well have written it yourself. If you pay me for a program that I let you believe correct, but it misleads you into misdirecting a client, who should be held responsible?

Imagine a courtroom scene wherein four of us are embroiled in a lawsuit brought, despite customary disclaimers, by your client. The manufacturer of your computer is the fourth party.

In my defence I prove that, on all reasonable computers, my program copes properly with all data in a reasonable domain and delivers at least half as many correct leading significant figures as the computer carries. You prove that your input data is reasonable and the output, though wrong, so plausible that you had no reason to withhold it from your client, who would have been happy with results half as accurate as I promised. The computer manufacturer's testimony affirms conventional wisdom: First, my program is defective because it uses algorithms generally regarded as "Numerically Unstable" and fails to take account of the computer's *special features* . Second, you are remiss for using hardware and software less accurate than you should have known you needed and could have bought. The judge is baffled by expert testimony; whom will he blame?

All the testimony in this scenario could be true. Lest you think a contradiction must lurk in it somewhere, here is an example drawn from [3] and designed to undermine faith in the foregoing kind of conventional wisdom. A program is needed to compute a polynomial $f(x)$ of degree 504 defined by composition thus:

$h(y) := (1/3 - y)*(3 + 3.45*y)$;
$g(z) := 1 + z + z^2 + z^3 + \ldots + z^{125} + z^{126}$;
$f(x) := g(h(x^2))$ for all $|x| < 1/\sqrt{3}$.

The program must run fast, the faster the better.

My program runs fast because it computes

$g(z) := (1 - z^{127})/(1 - z)$ if $z \neq 1$,
$:= 127$ otherwise.

On machines whose arithmetic is decimal (or hexadecimal, but not binary) I save space and time by omitting to test whether $z = 1$; since rounding 1/3 to 0.3333...3333 guarantees that $z := h(y) < 1$ for all $y := x^2 \geq 0$, I know $g(z) := (1 - z^{127})/(1 - z)$ is always safe.

When z is very close to 1 my program may look like just another fast way to calculate not $g(z)$ but *Junk* := *Roundoff/Roundoff* . However, tests reveal and proof confirms that my program cannot lose more than about half the significant figures carried on any machine whose every rational arithmetic operation introduces into its last significant digit delivered no more error than if the result had been chopped or correctly rounded or even rounded up by as much as 0.9 of a unit in its last digit. The program works correctly regardless of whether z^{127} is calculated by repeated squaring thus ...

$z^2 := z*z$; $z^4 := z^2*z^2$; $z^8 := z^4*z^4$;
$z^{16} := z^8*z^8$; $z^{32} := z^{16}*z^{16}$; $z^{64} := z^{32}*z^{32}$;
$z^{127} := z*z^2*z^4*z^8*z^{16}*z^{32}*z^{64}$; ... ,

or from the formula $z^{127} := \exp(127*\ln(z))$ used by many calculators, provided exp and ln suffer no worse error than my program allows for each rational operation. Since it does not need "correctly rounded" arithmetic, my program runs properly on IBM 370's and early DEC PDP-11's as well as on machines that round very carefully, as do DEC VAX's and recent H-P machines and those that conform to the rigours of the proposed IEEE floating-point standards p754 and p854.

But my program fails on CDC Cybers and UNIVAC 1108's and TI calculators, among others. Here is a table reporting results from a sampling of machines that perform only decimal arithmetic:

Names of Calculators	Sig. Dec. carried	Calculated f(0)
hp-10C,11C,12C,15C,16C,19C,22 27,29C,31E,32E,33E/C,34C 37E,38E/C,41C,67,91,92,97	10	127.00
hp-75,85,86,87	12	127.000
hp-21,25,35,45,55,65	10	127. *
Commodore SR4148R	12	127. *
hp-80 Financial	10	13.
TI Business Analyst, SR-30,40	11	100.
Commodore SR4190,5190	12	12.
Commodore SR1400, TI-MBA	12	0/0 Error
TI SR-52,56,51-II	12-13	128.
TI SR-50,50A,51,51A,58,58C,59	13	14.
Monroe 326	13	12.
VisiCalc 1.10 on the IBM PC	12	114.

The two entries marked * are the right answers for the wrong reasons, not proof of arithmetic quality.

Evidently this computation's accuracy depends not just on how many figures are carried but also on the manner in which figures are discarded. But the results seem to cry out for a value judgment: Faulty *Brand X* calculators? Or a pathological program rigged to cast undeserved aspersions?

I admit that, on all computers, my program is less accurate and not a lot faster than others that compute $g(z)$ from expressions like $(1+z)(1+z^2)(1+z^4)(1+z^8)(1+z^{16})(1+z^{32})(1+z^{64})$. Similar schemes work for $g_n(z) := (1 - z^n)/(1-z)$ when n is an arbitrary integer instead of 127, though they are not so obvious; one such scheme figures in financial calculations in the portable work-sheet computer "WorkSlate" just introduced by Convergent Technologies Inc. When n is not an integer the problem becomes truly interesting; see [3] and [6]. But the possibility that $g(z)$ might be computed on all machines by some other scheme better than my short program, even if no better scheme were visible yet, inhibits fair-minded folks from uttering premature condemnation and distracts them from the important question:

If a simple program works and is proved mathematically always to work well enough on all but a few commercially significant computers, who should bear the onus of adapting it to the aberrant machines?

In the past, the onus has fallen mostly upon the owners of aberrant machines or upon the creator of the program, rather than upon the creators of aberrant arithmetics. The future is unlikely to be different.

For the present, our best defence against arithmetic anomalies is some awareness of how certain computers generate them. The arithmetic aberration most common among computers, the one responsible for most of the anomalies exhibited so far in this paper, arises when a digit is jettisoned prematurely from the right-hand side of an internal register during an arithmetic operation. For example, consider the subtraction $d := 1 - z$ carried to five significant decimals with $z = 0.99999$ but otherwise performed as four machines do it:

Styles:	correct	CDC 7600	TI 59	TI MBA
z =	0.99999	0.99999	0.99999	0.99999
1 =	1.0000	1.0000 00000	1.0000	1.0000
z -> Z =	0.99999	0.9999 90000	0.9999_	1.0000_
1-Z =	0.00001	0.0000 10000	0.0001_	0.0000_
->	0.00001	0.0000 ____	0.0001	0
d =	1.0_{10}-5	0	1.0_{10}-4	0

Digits dropped prematurely have been replaced by underscores _ .

CRAYs' and UNIVAC 11XXs' subtractions resemble in binary the TI 59's in decimal. CDC's Cyber 205 differs from all the above; it may allege $z - 1 = 0 \neq 1 - z$. Although these disparities seem perverse, they are no worse than if either $1.00009 - z$ or $0.99999 - z$ replaced $1 - z$. Combining this insight with the mantra "*Backward Error-Analysis*" sometimes allays indignation, but not mine; for more on that subject see [6].

Premature abandonment of a digit defiles other arithmetic operations too. Multiplication is neither commutative nor monotonic on the TI 59 ; try $e \pi - \pi e$. Division on the TI Business Analyst gets a different quotient for 1/3 than for 9/27 . Double precision division in BASIC on the IBM PC alleges often that $X/1 \neq X$, and $1.000...0000 / 1.000...0001 \geq 1$.

After learning how these things happen, we can learn to look out for them and program around them, though they impose a deadening burden upon mathematical thought. To lift that onus from all of us, we must persuade the designers and builders of computer arithmetics that ...

1: aberrant designs can invalidate certain familiar calculations performed by most other machines without any trouble;

2: to compensate for aberrant arithmetic, software must become more complicated, costly and unreliable; and

3: their customers are aware of these truths. (I am not quite sure about item 3.)

THE AREA OF A TRIANGLE: Here is a familiar and straightforward task that blows up when subtraction is aberrant: Devise a program to compute the area $A(x,y,z)$ of a triangle given the lengths x, y, z of its sides. The program below will perform this calculation almost as accurately as floating-point multiplication, division and square root are performed by the computer it runs on only provided the computer's subtraction is free from the anomalies mentioned above. Consequently the program works correctly, and provably so despite roundoff, on an extremely wide range of machines:

APPLE III Pascal but not BASIC; Burroughs B6500 single precision; DG MV8000; DEC PDP-11 and VAX, and 10 and 20 single precision; ELXSI 6400; H-P 3000, 9000, 9836, 85-87, and all handheld machines except the hp-80; Honeywell 6000; IBM 370 and imitators, and rec. t IBM PC BASIC and FORTRAN; INTEL 8087, 86/330, 432; National 16081; recent PRIME machines; ZILOG S8000; ...

But the program miscalculates the areas of some needle-shaped triangles on those machines that discard a digit prematurely during subtraction. Among those egregious machines are ...

CDC Cybers and 7600; Cray 1; early IBM PC BASIC; early PRIME in double precision; TI calculators; UNIVAC 1108 and successors; ...

Of course, for each of those machines a method can be found to compute $A(x,y,z)$ as accurately as you like; but if the program must use only the machine's native floating-point equipment then nobody knows a fast program that can be proved to work on all machines, egregious or not. The classical formula due to Heron of Alexandria, namely $A(x,y,z) = \sqrt{(s(s-x)(s-y)(s-z))}$ where $s = (x+y+z)/2$, is numerically unstable for needle-shaped triangles regardless of whether every arithmetic operation is correctly rounded. For example, here is an extreme case worked out carrying just five significant decimals:

Given are x := 100.01, y := 99.995, z := 0.025. Then s := (x+(y+z))/2 = (200.03)/2 = 100.015 must round to either S := 100.01 or S := 100.02 to five sig. dec. Substituting S for s in Heron's formula yields either A = 0 or A = 1.5813 respectively, not the correct A = 1.000025... .

Evidently Heron's formula could be a very bad way to calculate, say, ratios of areas of nearly congruent needle-shaped triangles.

A good procedure, numerically stable for all but egregious machines, is the following:

```
Sort  x, y, z  so that  x ≥ y ≥ z  ;
If  z < x-y  then no such triangle exists ;  else
A := √((x+(y+z))*(z-(x-y))*(z+(x-y))*(x+(y-z)))/4
                ... DON'T REMOVE PARENTHESES! ...
```

How can so innocuous an algorithm fail on several egregious machines yet be provably successful on all the rest? Success depends upon the following easily proved ...

Theorem: If p and q are represented exactly in the same conventional floating-point format, and if $1/2 \le p/q \le 2$, then p - q too is representable exactly in the same format, unless p - q suffers exponent underflow.

(We shall ignore exponent over/underflow here lest its complications, which are avoidable, needlessly distract us from our discussion of roundoff problems; besides, p - q cannot underflow in arithmetic conforming to the latest drafts of IEEE p754 and p854 .)

The theorem merely confirms that subtraction is exact when massive cancellation occurs. That is why each factor inside $\sqrt{(\ ...\ )}$ is computed correct to within a unit or two in its last digit kept, and A is not much worse, on computers that subtract the way most people expect them to. Egregious machines do much worse; they miscompute some of the differences the theorem says they could calculate exactly. Watch what happens again in arithmetic to just five sig. dec.:

Styles:	correct	CDC 7600	TI 59	TI MBA
y =	99.995	99.995	99.995	99.995
x =	100.01	100.01	100.01	100.01
y →	099.995	099.99 50000	099.99_	100.00_
x-y →	000.015	000.01 ____	000.02_	000.01_
z =	0.025	0.025	0.025	0.025
z-(x-y) →	0.010	0.015	0.005	0.015
...				
A →	1.0000	1.1456	0.74997	1.1457
as if x →	100.01	100.00$\underline{5}$	100.01$\underline{5}$	100.00$\underline{5}$

Digits dropped prematurely have been replaced by underscores _ .

So, some procedure better than the "good" one above is needed to calculate reliably ratios of areas of nearly congruent needle-shaped triangles on egregious machines. Programmers, powerless to change these machines and reluctant to write a different program for each of them, might seek another "better" algorithm that works on all egregious machines as well as the rest. No such algorithm is known. My closest approach to it replaces every instance of a subtraction like p - q by a call to a programmed function Diff(p,q) designed to compute a satisfactory difference on all machines whether they jettison digits prematurely or not. Here is my attempt:

```
Real Function  Diff(y,x):  ... = y-x  with adequate accuracy.
Real values  y, x ;  real  d, e ;
If  |y| < |x|  then begin  d := -x ;  x := -y ;  y := d  end;
                ...  now  |y| ≥ |x| .
e := |x| ;
While  signum(x) = signum(y)
    do begin  d := 0.53*y ;  d := y - d ;
                ... DON'T do  d := y - 0.53*y  !
              x := x - d ;  y := y - d
       until  |y| ≤ e  endwhile;
Return  Diff := y - x  end Diff.
```

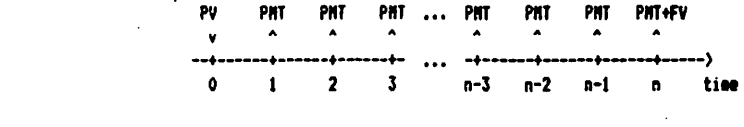
I believe this program works on *all* computers built in North America with hardware floating-point, egregious or not, *except* the CDC Cybers 203 and 205 and maybe some old WANG machines. I doubt that it works with every implementation of floating-point in software. I believe the multiplication by a magic number near 0.53 is unavoidable, and so is the necessity for a loop somewhat like the "While ... do ... until ..." loop in this program. And when it does work, how shall we decide which adds and subtracts in other programs to replace by calls to Diff ? If a program like Diff is the cure, the disease must be horrible.

In general, calculations near the singularities of functions of several variables are tricky at the best of times, so much so that they are described in pejorative terms, like *degenerate*, *ill-conditioned*, *ill-posed* and *unstable*, that tend to rub off onto whoever has to cope with them. My dismay at the way anomalous arithmetic makes the trickiest calculations trickier, often trickier than I can handle, is not shared by people who seem to think that only perverse calculations can be affected adversely, not the everyday world of dollars and cents. For their edification I turn now to dollars and cents.

FINANCIAL CALCULATORS: Four of these, the hp-92, -37E, -38C and -12C , are used now by several hundred thousand people to perform calculations concerning loans, leases, mortgages, sinking funds, annuities, amortisation schedules, depreciation, bonds, notes, net present value and internal rate of return of investments, and *Truth in Lending* regulations, among other things. The calculators were microcoded principally by Roy Martin [7], Dr. Dennis Harms [8] and Rich Carone, with some help from me to overcome mathematical difficulties. Businessmen are oblivious to these difficulties; to cope with, say, mortgages they need understand only the legends on five keys:

[n]	the number of periods, typically months.
[i]	the periodic interest rate, entered as a percentage.
[PV]	the Principal Value of the mortgage at the start.
[PMT]	the amount of each of n equal periodic PayMenTs paid at the End of each period. ([BEGIN] and [END] are keys too.)
[FV]	the Final Value, or "Balloon Payment", remaining to be paid at the end of the n^{th} period.

The signs of the cash-flows PV, PMT, FV tell us their directions, positive for incoming and negative for outgoing. With this sign convention in mind, the businessman visualizes the sequence of cash-flows in a mortgage transaction thus:



The same picture, but with different signs, depicts a sinking fund with initial deposit PV , n regular payments PMT , and an accumulated final value FV . The businessman need not know the equations that both transactions satisfy:

$$(1+x)^n\,PV + g_n(1+x)\,PMT + FV = 0 \quad \text{where} \quad g_n(z) := (1 - z^n)/(1-z) \quad \text{and} \quad x := i/100 .$$

(The troublesome function $g_n(z)$, with its removable singularity at $z = 1$, has appeared earlier in this paper with $n = 127$.)

Financial calculators are designed to solve these equations for any one of the five variables n, i, PV, PMT, FV given values for the other four. At first sight this task seems nontrivial only when the unknown is i , in which case a polynomial equation of degree n must be solved; n can be huge. Actually, the task must pose some challenge regardless of which variable be unknown, as the next example will show.

A Penny for your Thoughts.

A bank retains a legal consultant whose thoughts are so valuable that she is paid for them at the rate of a penny per second, day and night. Lest the sound of pennies dropping distract her, they they are deposited into her account to accrete with interest at the rate of 10% per annum compounded every second. How much will have accumulated after a year (365 days) ?

Enter data:

n :=	60*60*24*365	= 31,536,000 sec. per year.
i :=	10/n	= 0.000 000 317 097 9198 % per sec.
PV :=	0	
PMT :=	-0.01	= one cent per sec. to the bank.

Pressing [FV] should display one year's accretion but different financial calculators display different amounts:

Calculators	FV displayed
27, 92, 37, 38, 12	$ 331,667.00$_{47}$
BA	*293 539.16*$_{039}$
MBA	*334 858.18*$_{373}$
58, 58C, 59	331 559.38$_{38840}$

The small digits are not normally displayed, but are here to indicate how many figures the machines carry.

Why is the best result displayed by the machines that carry the fewest significant digits (10) in their data registers? Observing that erroneous results have lost more than half the figures carried, we should suspect that certain machines have subtractions and/or logarithms rather less accurate than the programmers of their financial procedures expected; and tests confirm our suspicions. Besides the anomalous subtractions uncovered above, we find that ln(0.9999995) is miscalculated on those machines as $-5_{10}-7$, not the correct $-5.00000125_{10}-7$, despite that they carry more than ten sig. dec. However, the owner of such a calculator might not be so suspicious at first; later he might check the consistency (but not the accuracy) of a result by treating it as a datum and back-solving for some other datum. For instance, recalculating i displays this:

Calculators	press [i] and see ...
27, 92, 37, 38	0.000 000 317*100*
12	0.000 000 3197*4*
BA	*catatonia*
MBA	0.000 000 *3886*
58, 58C, 59	0.000 000 3154

If their accuracy is not impressive, yet their speed is worth a thought; while performing fewer than about a dozen floating-point operations per second, most of these machines take less than one or two dozen seconds to solve a polynomial equation here of degree $n = 31,536,000$. We shall return to this thought.

A single somewhat artificial sample is not enough to demonstrate how much the probability of computational failure is inflated by anomalous arithmetic. But before drawing further samples, we should digress to reconsider the significance of "artificial" examples.

Equation-solving is an iterative process akin to exploration. Regardless of how typical the data and solution may be, the path followed by the iteration from first guess to final result may approach or enter regions that are financially implausible though mathematically legitimate and still informative. Therefore, programs that do not allow an equation to be evaluated accurately over the widest domain on which it makes sense mathematically must cramp an equation-solver's style, as the next example shows.

Yield from a Risky Investment.

For an investment of $-PV := \$\ 35,000,000$ now, investors are promised $n := 100$ equal monthly installments of an amount PMT yet to be agreed upon, but between $\$\ 640,000$ and $\$\ 1,000,000$, plus a final payment at the 100^{th} month of $FV := \$\ 100,000,000$. How does the yield i, reckoned in % per month, vary with PMT ?

Tabulated in the first column below are selected values of PMT, with the corresponding yield in the second column shown as displayed on any of the hp-92, -37E, -38C or -12C after about a dozen seconds of calculation. The third column shows what the TI MBA displayed.

PMT	true i %	i on the MBA
$ 640,000	2.314053	2.314053
650,000	2.335758	-1_{E}-97 *BLINKING*
660,000	2.357528	2.357528
800,000	2.669065	2.669065 after a long time.
1,000,000	3.135506	-2106.949 *BLINKING*

The blinking tiny number is a symptom of roundoff troubles. The other anomalies could be caused by an unfortunate choice of iterative method for the equation to be solved.

SOLVING EQUATIONS: The customary iteration for solving any given equation $f(x) = 0$ is

Newton's iteration:

$x_{k+1} := x_k - f(x_k)/f'(x_k)$ for $k = 0, 1, 2, \ldots$

starting from a suitable first guess x_0. If it converges, the iteration normally converges quickly, ultimately nearly doubling the number of correct figures with each iteration, so that high accuracy does not cost very much. But the financial equation above is abnormal because, though a polynomial equation in $x = i/100$, its degree n can be so huge that the graph of the polynomial is, for practical purposes, spiked and/or stepped rather than smooth. Consequently, Newton's iteration converges too slowly if it converges at all. At first sight, the following lemma makes the situation appear hopeless.

Lemma: Newton's Iteration is Ubiquitous; if X is a continuous real function and if the iteration $x_{k+1} := X(x_k)$ converges, from every starting point x_0 sufficiently close, to a root of the equation $F(x) = 0$, then the iteration must be Newton's iteration applied to an equation $f(x) = 0$ equivalent to $F(x) = 0$ in the sense that both have the same root.

The proof, using $f(x) = \exp(\int dx/(x-X(x)))$, is easy. The lemma tells us not to bother trying iteration to solve an equation unless it can be transformed into an equivalent one well suited to solution by Newton's iteration. What does "well suited" mean? One meaning I discovered is this:

Theorem: If $f(x)$ is a difference $f = u - v$ between two convex functions, one monotone nondecreasing and another monotone nonincreasing throughout some real interval, then Newton's iteration $x_{k+1} := x_k - f(x_k)/f'(x_k)$ cannot dither; it must either escape from that interval or converge within it, no matter where therein the iteration starts.

This, the most general sufficient condition known for the convergence of Newton's iteration applied to solve a real equation, was not easy to prove, but it was worth the effort. The financial equation above, when it has just one financially meaningful solution i, can always and easily be transformed into the form

$$\ldots + c_{-3}y^{-3} + c_{-2}y^{-2} + c_{-1}y^{-1} + c_0 = c_1 y + c_2 y^2 + c_3 y^3 + \ldots$$

where each c_j is the magnitude of a cash-flow and y is either $1+x$ or $1/(1+x)$, whichever ensures that $c_0 > 0$. This form satisfies the theorem throughout the interval $x > -1$, capturing all interest rates $i > -100$ % ; no others make financial sense. Now, applied to the transformed equation, Newton's iteration must converge from every starting point. But not very fast if n far exceeds 1000.

To cope with huge n on the hp-92, Roy and I approximated the root x of the financial equation asymptotically (as $n \to \infty$), and used the leading term as a first guess for the iteration. Despite having to recognize several cases, the approximation is quick and, when n is large enough that it matters, accurate to over five sig. dec. Therefore, nobody has to wait more than about a dozen seconds, long enough for fewer than 100 multiplications, after pressing [i] on the hp-92, -37E, -38E or -38C, no matter how big n may be.

Dennis and I used related transformations to solve related equations for Internal Rates of Return on the hp-38E and C, whose [IRR] key will cope with over 2000 cash-flows. Later, to cope with a revised version of the financial equation above that, unlike the original, makes sense when n is not an integer, Rich and I used yet another transformation in the hp-12C ;

we used $\ln(y)$ instead of y as the independent variable in the equation above with terms $c_j y^j$, and applied Newton's iteration to its logarithm. Although each iteration cost now more time than before, the theorem continued to guarantee convergence which was rapid from every starting point regardless of n. Further details are not needed to make my point:

Every day, hundreds of thousands of people employ powerful financial calculators that are convenient, fast and reliable because of Physical, Chemical, and now Mathematical technology more intricate than they imagine.

Euphoric at the success of the hp-38E, Dennis Haras' manager, Stan Mintz, humoured us by granting permission to devise a calculator with a [SOLVE] key, despite that no marketing survey had revealed any demand for such a thing, and subject to one proviso: mindful of his struggles with integrals in college, he charged us to devise an [INTEGRATE] key too. Thus was the hp-34C born. Its innovations have been exposed elsewhere [9,10], but not the mathematical insight that made a [SOLVE] key seem feasible. Here is the train of thought ...

Suppose we are given an equation $f(x) = 0$ to solve but not much time to study it. Suppose we are willing to try Newton's iteration, perhaps because the Theorem above is applicable or for lack of a better idea. We will have to write a program to compute $f'(x)$ as well as $f(x)$, unless we choose to approximate the derivative by a difference quotient. This choice is tantamount to approximating a tangent by a secant, whence the iteration formula gets its name, i.e.

Secant Iteration:

$$x_{k+1} := x_k - f(x_k)(x_k - x_{k-1})/(f(x_k)-f(x_{k-1})) .$$

If this iteration converges, then it is known to converge usually slightly faster than Newton's unless calculating $f(x)$ and $f'(x)$ together costs less than about 45 % more time than calculating $f(x)$ alone. But will the secant iteration converge? More to the point, will the approximation of a tangent by a secant leave intact whatever reasoning might have motivated recourse to Newton's iteration? Almost surely YES! More precisely, I discovered the following

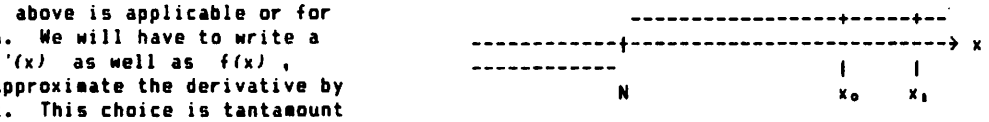
Phenomenon: Suppose that Newton's iteration to solve the equation $f(x) = 0$ converges from every starting guess within an interval to a root therein. Then, unless $f(x)$ vanishes inside that interval without reversing sign there, the secant iteration must converge to the same root from every pair of starting guesses in that interval.

The proof that this must happen is extremely long and difficult partly because $f(x)/f'(x)$ could oscillate pathologically in the neighbourhood of a root where both $f(x)$ and $f'(x)$ vanished simultaneously. The phenomenon's implication is immediate; the Secant Iteration provides as firm a foundation as Newton's for a general-purpose equation solving program, but with no need for a derivative. So we created such a program [9], and Tony Ridolfo microcoded it into the hp-34C under the [SOLVE] key with no scratch registers to spare. Later the same program was copied into the hp-15C. To use it to solve $f(x) = 0$, follow these three steps:

- Enter into the calculator under, say, label [A] a program that evaluates $f(x)$ given any x. (Other labels can be used instead of [A].)
- Enter a guess or two at the desired root, the closer the better.
- Press [SOLVE] [A] and see what happens ...

If $f(x)$ changes sign anywhere, then [SOLVE] will surely locate such a place to within a few units in its tenth sig. dec. whenever ...
-- $f(x)$ is strictly monotonic, or
-- $f(x)$ is convex, or concave, or
-- $|f(x)|$ has no nonzero local minimum, or
-- $f(x)$ has different signs at two guesses.
If both the last two conditions are violated, then [SOLVE] may display an approximation to the location of a nonzero local minimum of $|f(x)|$ and signal that it could not find a change of sign. Under no circumstances will [SOLVE] run indefinitely; it always finds something, even if sometimes the search takes a long time. Here is an example:

$$B_N(x) := \text{signum}(x-N) = \begin{cases} +1 & \text{if } x > N, \\ 0 & \text{if } x = N, \\ -1 & \text{if } x < N. \end{cases}$$



Try, say, $N := 7$ and first guesses $x_0 := 101$ and $x_1 := 102$. The program for $B_7(x)$ is this:

```
LBL B  7  -  ENTER  ABS  x≠0?  ÷  RTN
```

To enter the first guesses and solve $B_7(x) = 0$ for x, press 101 ENTER 102 SOLVE B and wait a minute to see $x = 7.000000000$ displayed after $B_7(x)$ has been sampled 45 times. (How does [SOLVE] know which way to turn? See [9].) Changing N from 7 to 0 extends the time to 6 min. after 361 samples. Yet longer search times in difficult cases might have been lessened had a few more than the five scratch registers allocated to [SOLVE] been available in the hp-34C, but [INTEGRATE] consumed a lion's share.

THE [INTEGRATE] KEY: Among innumerable numerical quadrature procedures available in the literature and in computers, what distinguishes this one is its relative ease of use. Estimating

$$I := \int_y^x f(t)\, dt$$

on the hp-34C and hp-15C entails these steps:

- Enter into the calculator under, say, label [A] a program that evaluates $f(x)$ given any x. (Other labels can be used instead of [A].)
- Set the display to show as many digits of the integrand f as matter. (More on this below.)
- Put in the limits of integration thus:
  y ENTER x
- Press [$\int_y^x$] [A] and wait for the results. ...

Foremost in the display, in the X-register, will be the estimate of the desired integral I ; behind it, in the Y-register, will be the uncertainty δI in I inherited from the tolerance allowed in f . More precisely, the [$\int_{\Sigma}^{\Sigma}$] key estimates not merely $I = \int_a^b f\,dt$ but actually $I \pm \delta I = \int_a^b (f \pm \delta f)\,dt$ where all that is asserted about δf is that $f(t) \pm \delta f(t)$ agrees with $f(t)$ in all digits displayed. Geometrically, the graph of $f \pm \delta f$ is a ribbon, centred along the graph of f , containing all graphs regarded as practically indistinguishable from that of f . The area under the graph of f is I , and is uncertain by $\pm\delta I$ where $2\delta I$ is the area of the ribbon.

Here is a familiar example:

$$I := Q(x) := \int_x^{\infty} \exp(-t^2/2)\,dt/\sqrt{(2\pi)} \ .$$

Since the integrand underflows past 10^{-99} to zero when $t > 22$, replacing the upper limit ∞ by 22 discards nothing but converts the improper integral $Q(x)$ into a proper one that any general purpose numerical quadrature program can evaluate easily. Designate this procedure "Method A" ; as we shall see, it will waste most of its time sampling the integrand at places where it contributes negligibly to the integral. Another procedure, designated "Method B" , substitutes $s^2 = \sin^{-1}(\exp(-t^2/2))$ to transform the improper integral $Q(x)$ into a proper integral:

$$Q(x) = \int_0^{\sqrt{(\sin^{-1}(\exp(-x^2/2)))}} s\,\frac{\sqrt{((\sin(s^2)+1)(\sin(s^2)-1)/\ln(\sin(s^2)))}}{\sqrt{\pi}}\,ds \ ,$$

except if $x < 0$ calculate $Q(x) := 1 - Q(-x)$. Although the transformed integrand is finite everywhere, it does have two weak singularities:

One is at $s = 0$ where an attempt to calculate ln 0 could stop the calculator, but it won't; the [$\int_{\Sigma}^{\Sigma}$] key is designed to avoid drawing samples of the integrand from the ends of the range of integration lest singularities that are otherwise easily integrable derail it there.

The second is a removable 0/0 type singularity that occurs when $s^2 = \pi/2$. It looms near when x is so tiny, and the upper limit of integration so nearly $\sqrt{(\pi/2)}$, that s^2 approaches $\pi/2$ near enough for $\sin s^2$ to round to very nearly 1 ; then both $\sin s^2 - 1$ and $\ln \sin s^2$ will be seriously contaminated by rounding error.

Could that error reduce the integrand to useless *Junk* := *Roundoff*/*Roundoff* ? Not on the hp-34C nor hp-15C. The roundoff cancels itself; treat $\sin s^2$ instead of s as the independent variable to see why. Therefore, the integrand will be evaluated accurately provided subtraction and logarithm are both accurate to full working precision, as they are on these machines but not some others.

The programs for methods A and B are short enough to show here:

```
LBL A  22  ∫1  π x↕y  LBL 0  f÷  X↕Y LSTX ÷ X↕Y RTN
  LBL 1  X↑2 2 CHS ÷ exp RTN
LBL B  RAD  GSB 1  SIN⁻¹ f√ 0 X↕Y ∫2  π  GTO 0
  LBL 2  ENTER X↑2 SIN LN LSTX 1 - ÷ LSTX 2 ÷ X↕Y ÷ f√ RTN
```

Before they are run, the display should be set to show just as many figures as are wanted. For four significant figures, press [SCI] 3 . Shown below for both methods and for a few values of x are estimates of the integral $Q(x)$, and how often the integrand was sampled to get each, and the elapsed time.

x	Q(x) by Method A and by Method B	# samples # samples	sec. sec.
10	$7.6198_{10}\text{-}24 \pm 17_{10}\text{-}28$	127	227
------	$7.6199_{10}\text{-}24 \pm 18_{10}\text{-}28$	7	27
1.96	0.024998 ± 0.000006	127	227
------	0.024998 ± 0.000006	15	58
0	0.499999 ± 0.000045	63	116
------	0.500003 ± 0.000146	15	58

For higher accuracy, say 7 or 8 sig. dec., press [SCI] 7 before running the programs; typical results for methods A and B respectively are

$Q_A(0) = 0.4999999998 \pm 0.0000000047$ at 255 samples in 444 sec.
$Q_B(0) = 0.5000000002 \pm 0.0000000135$ at 63 samples in 216 sec.

This example makes it all seem easy. Actually, reliable and rapid numerical integration is still somewhat a black art, especially when combined with devious transformations to tame otherwise wild or nearly improper integrals. Frequently these transformations flirt with singularities. Some singularities, designed to cancel each other harmlessly, will do just that despite roundoff . because the underlying arithmetic and elementary functions in the hp-34C and hp-15C have been implemented so carefully. Other singularities cannot be removed but can be weakened enough to be tolerated by the [$\int_{\Sigma}^{\Sigma}$] key's quadrature procedure [10] ; and then even if thousands of samples of the integrand have to be accumulated they will be added so accurately, because the calculators carry three extra digits for the purpose, that roundoff inside the quadrature procedure will not obscure the desired result.

The user of these machines can remain blissfully unaware of details that, on some other computers, could bring grief to a program he thought was pretty clever.

However, no integration procedure nor equation solver based exclusively upon a sampling strategy can be foolproof. To understand why, consider a procedure that purports to accomplish one of the following tasks for an arbitrary function f given only a program that calculates $f(x)$ for any given argument x in some specified range:

-- Evaluate $\int f(t)\,dt$ over the given range.
-- Minimize $f(x)$ over the given range.
-- Find out whether and where $f(x) = 0$.

We shall test this procedure first upon a program that returns always $f(x) := 1$ but also prints out a record of its argument x . Then for some finite N we shall know that the procedure drew samples $f(x_1)$, $f(x_2)$, $f(x_3)$, ..., $f(x_{N-1})$, $f(x_N)$ while attempting to accomplish the assigned task. Next let us test the procedure upon a second program that returns

$$f(x) := 1 - (c(x-x_1)(x-x_2)(x-x_3)(\ \dots\ )(x-x_{N-1})(x-x_N))^2 \ ,$$

where c is chosen so big that f reverses sign more than twice. Since both functions f(x) return exactly the same value 1 for every sample drawn, the procedure must deliver the same result for both functions; but no such result can be correct for both.

Therefore the [∫x̄] key must be as fallible as all other sampling procedures. Spikes or jumps or violent oscillations can precipitate failure. For example, attempts to evaluate numerically $\int_{0.01}^{0.1} (t-0.05)\, t^{10} \exp(1/t)\, dt = -134.26994...$ too often deliver instead a very wrong estimate like +0.1359 . That is the area under the graph of the integrand between t = 1 and t = 0.6 , an area shaped like a triangular sail. The graph practically coincides with the t-axis between t = 0.6 and t = 0.016 . Between t = 0.016 and t = 0.01 the graph is a sharp spike rising from -1,075,246.9 at t = 0.01 up to -1.571 at t = 0.012 , up to -0.0106 at t = 0.013 , and nearly zero thereafter. Therefore, most of the integral lies in a narrow spike only 1/500th the width of the range of integration. Sampling is most unlikely to reveal that spike unless the samples are very numerous, as is the case only when high accuracy is desired. Evaluating the integral in the obvious way with 3 sig. dec. displayed ([SCI] 2) on the hp-34C yields the expected misleading result +0.1357 ± 0.0003 after 31 samples. With 4 sig. dec. displayed ([SCI] 3) the result is -134.26994 ± 0.02 after 2047 samples, correct but costly. A more economical way to evaluate this integral is as a sum $\int_{0.01}^{0.1} = \int_{0.8}^{1.8} + \int_{0.01}^{0.8}$; each term can be be evaluated separately and added later to yield -134.270 ± 0.022 after 126 samples all told at [SCI] 3 . Neither this partitioning of the integral nor its necessity would be obvious to someone who did not know what to look for; the [∫x̄] key could mislead an uneducated user badly.

THE CALCULATOR OWNER'S HANDBOOK: A computer is deemed *Reliable* when its users are never surprised by something its designers must later apologize for. How can designers and users who never meet learn what to expect from each other? Through education. That is the key to reliable computation. Exhorting manufacturers to build reliable equipment is mere counsel of perfection unless they can learn how to design it at a tolerable cost. And then, as refined equipment free from avoidable anomalies becomes available, users must be taught what to expect and how to exploit it. Obviously, expectations will be influenced, if not taught , by the Owner's Handbook and whatever other documents the manufacturer supplies to inform and indoctrinate the customer. Communication the other way is less obvious; only recently have some manufacturers come to appreciate how much they learn from the Owner's Handbook before it is written, before the machine is designed.

How should arithmetic be designed? A simple goal for most of a calculator's arithmetic functions would be easy to state [11] :

> Keep the error strictly smaller than one *ulp*.
> (An *ulp* is one *U*nit in the *L*ast *P*lace.)

But this specification accomplishes less than one might reasonably desire; for instance it ensures neither the sign-symmetry of $\sin(x) = -\sin(-x)$ nor the monotonicity of $\sqrt{x}$. Neither is the goal easy to achieve; sometimes it is impractical. For example, recent hand-held Hewlett-Packard calculators that accept and deliver data to 10 sig. dec. produce two results,

$729^{33.8} \rightarrow 7.968419666_{10}95$ and $3^{201} \rightarrow 7.968419664_{10}95$,

of which at least one (it is the latter) must err by more than one ulp. Only near the overflow and underflow thresholds do the exponential functions go so far as two ulps wrong; to keep their error below one ulp here too would have required that intermediate calculations be carried to more than the 13 sig. dec. actually carried in a few internal registers of these machines. Would the cost and speed penalties paid to carry an extra figure be offset by noticeably enhanced accuracy? Not likely. And some offensive inaccuracies would persist even if twice as many figures were carried. Consider $\sin(\pi) = 0$. This equation presumes that the $\sin(...)$ procedure is given exactly $\pi = 3.14159\ 26535\ 89793\ 23846\ 26433...$. But, instead of π , the [π] key delivers [π] = 3.14159 2654 = π rounded to 10 sig. dec. ; only after we notice their difference will we recover from our initial surprise at pressing the [SIN] key and seeing [SIN]([π]) = $-4.10_{10}-10$ instead of zero. Our second surprise is finding error in the 4th instead of 10th sig. dec. of [SIN]([π]) ≠ sin([π]) = -0.00000 00004 10206 76153 7... . This gross error is due to the calculator's use internally of only 13 sig. dec. of π . Larger radian arguments incur larger errors;

[SIN]([π] $_{10}14$) = +0.79905 50814 ≠ sin([π] $_{10}14$) = -0.78387... .

(Angles in Degrees incur no such errors; for instance [TAN](10^k °) = -5.671281820 correctly for k = 2, 3, 4, 5, ..., 99 .) The only way to avoid such errors with large radian angles is to retain π to very high accuracy; over 120 sig. dec. would be needed for these calculators. That extravagance is feasible and attractive in large computers with large memories [12] , but not in calculators. Besides, because uncertainties so small as half an ulp in the input arguments swamp the errors we have been discussing, these errors have almost no impact upon the scientific and engineering calculations for which calculators were designed. What little impact might remain is further attenuated by the preservation, to within an ulp or two on these machines regardless of how big x may be, of identities like $\sin(2x) = 2\sin(x)\cos(x)$ that do not involve π explicitly. Therefore errors caused by not using exactly π , and the convoluted excuses for them, are tolerable; for more details see [6].

Intolerance would not simplify the situation much. Suppose we insisted upon Perfection and found it,- a machine whose every arithmetic function rounds correctly to within half an ulp. (This is feasible for algebraic functions but impractical for exponential and transcendental functions.) Would this Perfection preclude arithmetic surprises? Regardless of the breadth of our experience, NO . For example, many an inexperienced calculator user would continue to

be surprised that $(\sqrt{x})^2 = x$ is often spoiled by roundoff; on decimal machines violations abound for $1 < x < 10$ and $25 < x < 100$ but none lie in $10 \leq x \leq 25$. On the other hand, experienced cynics, expecting nothing to survive roundoff, must be surprised to discover, on binary and quaternary machines but not on those with larger radix, that despite roundoff $\sqrt{(x^2)} = |x|$ for all x unless x^2 over/underflows. These surprises can be confirmed first by experiment, then by simple proofs. Recent results of Harry Diamond [13] suggest that surprises like these must pervade correctly rounded arithmetic. Yet something worse lurks there.

Correctly rounded arithmetic conceals anomalies so rare that no conscientious programmer could reasonably be expected to discover them. We do not expect such a programmer to prove his every program correct; doing so might entail a proof as difficult as that of the Four Colour Theorem for planar maps. Alternatively, the programmer might be forced to insert defensive code to cope with eventualities that almost never happen, if they can happen at all. Either way slows down the programmer; and defensive programming slows down the program too. Besides, whatever causes errors in programs also causes errors in proofs. Therefore every program must be run through tests upon sample data drawn reasonably densely from its domain. But some anomalies are too rare to be caught by that kind of test. For instance, consider a function $f(x) := x - \sin(x)$ that figures in problem 2 on p. 12 of P. Henrici's book [14]. $f'(x) = 2\sin^2(x/2) \geq 0$, so $f(x)$ must be monotone non-decreasing. Can the same be said for $F(x) := x - \mathrm{SIN}(x)$ where $\mathrm{SIN}(x)$ is $\sin(x)$ correctly rounded ? Yes, everywhere except at a scattered handful of exceptions, each an accident of radix and wordsize. For instance, when rounding to 6 sig. dec. the sole exception is at $x = 0.100167$; to 5 sig. dec. it is at $x = 0.010000$; to 4 sig. dec., nowhere:

x	$\sin(x)$	$\mathrm{SIN}(x)$	$F(x)$
0.100167	0.09999958095	0.0999996	0.0001674
0.100168	0.1000005759	0.100001	0.0001670
0.010000	0.009999833334	0.0099998	0.0000002
0.010001	0.01000083328	0.010001	0

So, uncompromising adherence to the most rigorous rules for approximate arithmetic will not protect a computer from unpleasant surprises. Apparently the approximation of the continuum by a discrete set must introduce some irreducible quantum of noise into mathematical thought, as well as into computed results, and we don't know how big that quantum is. If we have to tolerate this unknown noise, we might as well tolerate a little more. Tolerance grants the designer of a computer's arithmetic not *carte blanche* for arithmetic anarchy but rather his mandate:

Keep both noise and cost *tolerably* small, the smaller the better.

Tolerable to whom? To the customer, to whom the designer would rather not have to apologize for unfortunate consequences of a compromise that may have been unnecessary. Thus do we circle back to the real world, where Science can tell us how to do it, or not to try, but not what to do. The designer of computer arithmetic must be guided in his choices by something more than mathematics:

Design arithmetic functions in such a way that almost no user need know more about them than the designer is proud to explain in the Owner's Handbook.

If the handbook says nothing much about the accuracy of the functions, then they had better be so accurate that nothing much need be said. Such is the case for all financial functions and all elementary real functions of one or two real arguments on recent Hewlett-Packard hand-held calculators. Rational operations $(+, -, \times, \div)$ and $\sqrt{x}$ are correctly rounded to within half an ulp; the logarithms and inverse trigonometric and inverse hyperbolic functions are almost as good. No errors worse than the subtle ones shown above afflict trigonometric functions of radians, and exponential, hyperbolic and gamma functions. (The [x!] key delivers $x! = \Gamma(x+1)$ for non-integers on the hp-34C and hp-15C .) So little worse than best possible are these errors that no mention of them appears in the Owner's Handbook, though an auxiliary handbook describes them fully in a chapter [6] destined to be forgotten as soon as it is read. On the other hand the [Σ+] key, used to calculate standard deviations and perform linear regression upon pairs (x_j, y_j), uses algorithms chosen more for compatibility with past practice and for speed than for numerical infallibility, and gives unreliable results when *all* the data x_j agree in their first several sig. dec. The Owner's Handbooks supply a simple and efficient remedy: temporarily omit redundant leading digits. In other words, when all data are very close to their mean, subtract an approximate mean from them before entering them.

So far, the Owner's Handbook has been depicted as more a contractual than tutorial document. It tells the customer what he has bought, offering advice only when it is brief and necessary to avoid misunderstanding. The manufacturer of the computer is not obliged to teach the customer how to compute. That policy seemed sound until it collided with the hp-34C whose powerful [$\int$] and [SOLVE] keys invite abuse. Where would the customer learn how to use those keys reliably? Not from standard texts on Numerical Analysis ; they tend to drown the reader in formulas none of which match the calculator's algorithms. Hardly any text explains how to recognize wild integrals and tame them, or what to do when an equation-solving iteration finds no root. Whether these be rare pathologies or not, they must happen daily to at least several among the hundreds of thousands of users of the calculator. Where would blame for these pathologies come to rest?

Robert Barkan and Hank Schroeder wrote most of the Owner's Handbook for the hp-34C . They were not confident that they could reverse a long standing policy against tutorial matter in the handbook when they decided to include two extra chapters, one on integration and one on equation solving. Each chapter discusses its subject's pathologies with examples worked out on the calculator, but the discussion is otherwise

independent of the calculator's particulars; these chapters, like the subsequent articles [9] and [10], might well have been written for a text on numerical methods. The chapters constitute part of an appendix at the end of the handbook so that nobody will think he has to read them before using the calculator. Indications are that everyone who uses the [∫ᵡʸ] and [SOLVE] keys has read those chapters and appreciates them.

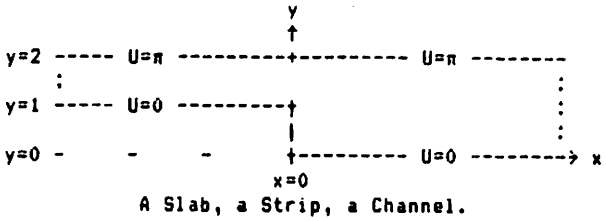
Something else was needed for the hp-15C . The user of this slim (128mm. x 80mm. x 15mm.) shirt-pocket calculator can, in a single keystroke, attempt to invert a singular matrix, or evaluate a complex analytic function at a slit-discontinuity in its domain. Tutorial chapters for this machine could amount to a text covering two years of college mathematics for engineers, leaving out only vector calculus (divs, grads and curls). Our inclination to embed such a text in the Owner's Handbook was deflected by a prudent marketing specialist who explained to us ...

"The Intimidation Factor:
A potential customer, wishing to purchase an advanced scientific shirt-pocket calculator, peeks into the box and sees nestled there a slim calculator beside a very thick book. ..."

Instead we put tutorial matter into a second book [15] that a calculator owner could buy later.

COMPLEX NUMBERS AND MATRICES: The hp-15C is distinguished from all previous calculators by its treatment of complex numbers and matrices as arithmetic objects in their own right [16] rather than as mere aggregates of numbers. The rational operation keys [+], [-], [x], [÷] and [1/x] act upon complex numbers or upon matrix operands just as they act upon real numbers; other keys like [$\sqrt{x}$], [y^x], [SIN], [$\cosh^{-1}$], etc. calculate their analytic functions of complex as well as real numbers. The [ABS] key delivers $|x|$ for real or complex x ; other key strokes deliver the determinant and various norms of a matrix. Of course, earlier calculators and computers can be programmed to perform similar operations, albeit not so easily nor so accurately. The hp-15C takes the tedium out of these operations; in a small package it offers some of Fortran's convenient handling of complex arithmetic, some of APL's convenient handling of array arithmetic. Teachers see more than mere convenience there; students using the hp-15C can experiment with powerful abstractions and learn their value before having to learn how to implement them.

To illustrate the value of convenient complex arithmetic, let us apply it to three problems in Mathematical Physics, all sharing the following figure in the (x,y) plane:



A Slab, a Strip, a Channel.

Problem 1. The figure shows the cross section of a large metal slab whose thickness doubles just as a straight line is crossed. The slab's flat upper surface is kept at a constant temperature $U = \pi$. The lower surface, with the step, is kept at a constant temperature $U = 0$. How does the temperature $U(x,y)$ vary inside the slab?

Problem 2. Material of uniform resistivity and thickness is laid down in a very long strip whose width doubles at the step shown in the figure. An electric current passes through the strip; how must the voltage $V(x,y)$ vary in the strip?

Problem 3. The figure looks down upon a long channel of constant depth whose width doubles at the step. Water flows slowly along the channel. Floating in the water is a tiny cork chip; what path must it follow? The path, a "stream line", is a level curve of a "stream function" $U(x,y)$.

$U(x,y)$ and $V(x,y)$ both satisfy the same partial differential equation, Laplace's equation

$$\partial^2U/\partial x^2 + \partial^2U/\partial y^2 = 0 = \partial^2V/\partial x^2 + \partial^2V/\partial y^2 ,$$

but with different boundary conditions. U takes boundary values shown in the figure. The normal derivative of V vanishes upon the boundaries shown in the figure, and V/x tends to a limit as $x \rightarrow \infty$ and to twice that limit as $x \rightarrow -\infty$.

Engineering students are usually taught a finite difference or finite element method to calcuate U numerically. A mesh is laid upon the strip to partition it into many tiny cells. To each cell corresponds an equation saying that U therein approximates a weighted average of its values in neighbouring cells. The solution of this system of equations approximates U . The usual way to improve accuracy is to refine the mesh, thereby increasing the number of equations to be solved. Because the solution U has a singularity at the intruding corner (at $x=0$, $y=1$) , it will not be approximated well near there unless the mesh near there *is* refined. Therefore, calculating U this way must be tedious. If Mathematics be the Art of Calculation without Computation, this is not Mathematics; it is more like Simulation.

Table 1: Points (x,y) on the Stream Line U = 0.01 .

U:	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
V:	-1.6	-1.2	-0.8	-0.4	-0.2	-0.1	0	0.1	0.2	0.4	0.8	1.2	1.4	1.6
x:	-0.3680	-0.2540	-0.1480	-0.0566	-0.0209	-0.0075	0.0002	0.0012	0.0018	0.0028	0.0053	0.0118	0.1367	0.5248
y:	1.0029	1.0028	1.0025	1.0020	1.0015	1.0011	1.0002	0.9921	0.9769	0.9307	0.7724	0.4636	0.0448	0.0130

The classical mathematical solution of the three problems employs complex variables and conformal transformation: Associate position in the plane with the complex variable $z := x + iy$ and let $w(z) := V + iU$ be composed from the solutions $U(x,y)$ and $V(x,y)$ of the three problems. Here $i^2 = -1$. Rather than express w in terms of z, we shall express z as a function of w, as is convenient for plotting level curves along which either U is constant or V is constant in the z plane. It turns out that

$$z = (2\cosh^{-1}((2e^{w}-5)/3) - \cosh^{-1}((5-8e^{-w})/3))/\pi .$$

To type this expression onto the page takes about twice as many keystrokes (72 vs. 38) as to enter the program that calculates it into the hp-15C :

```
LBL C  eˣ ENTER ENTER + 5 - 3 ÷ COSH⁻¹ ENTER +
       5 ENTER 8 R↑ ÷ - 3 ÷ COSH⁻¹ - π ÷ RTN
```

To plot stream lines, curves along which U is constant, use the program as follows. Choose a constant value between 0 and π for U. As V runs from -8 to $+4$, say, so does $w := V + iU$ run along a horizontal line segment in the w plane whose image in the z plane is the desired stream line. A point $z := x + iy$ on that curve is located by pressing keys thus:

KEYSTROKES	OPERATION PERFORMED	DISPLAY
V [ENTER] U [f] [I]	Create $w = V + iU$	V c
[f] [C]	Calculate $z = x + iy$	x c
[f] [(i)]	Display imaginary part	y c

(The annunciator "c" indicates when only one part, real or imaginary, of a complex value is being displayed.)

For example, Table 1 shows how the stream line $U = 0.01$ bends around the intruding corner. Each point costs about 15 seconds to calculate and plot on graph paper, so tedium has not been banished entirely; some time must pass before inexpensive shirt-pocket calculators will be able to display a plot of stream lines automatically. On the other hand, some time must pass before computers capable of driving graphics screens or pen-plotters can be expected to possess as full a set of complex elementary functions as has the hp-15C. Only recently have such functions begun to appear in a few APL installations. For over twenty years, full implementations of Fortran have included complex arithmetic too, but not all the elementary functions; for instance, $\cosh^{-1}$ is missing. Therefore, the formula for z above would have to be transformed by the little known substitution

$$\cosh^{-1}(q) = 2\ln(\sqrt{((q+1)/2)} + \sqrt{((q-1)/2)})$$

into something expressible in Fortran;

$$z = (2/\pi)\ln(\sqrt{(e^{w}/3)}\,(\sqrt{(e^{w}-1)}+\sqrt{(e^{w}-4)})^2 / (2\sqrt{(e^{w}-1)}+\sqrt{(e^{w}-4)})) .$$

Confirming this transformation requires, besides tedious algebra, careful analysis to check that it maps boundary values correctly. Such checking is nontrivial because familiar formulas valid for real functions frequently fail for the principal branches of multi-valued complex analytic functions [17]. For instance, formulas like

$\sqrt{(xy)} = \sqrt{x}\,\sqrt{y}$, $\ln(xy) = \ln(x) + \ln(y)$ and $\cosh^{-1}(x) = \ln(x + \sqrt{(x^2-1)})$, valid when x and y are real and positive, may fail when x and y are complex in the left half-plane. Moreover, rounding errors can ruin formulas that would be correct otherwise, as happens to the substitution for $\cosh^{-1}(q)$ above when q is near ± 1.

Formulas robust in the face of roundoff are hard to find; the following instance is used in the hp-15C to calculate $r + is := \cosh^{-1}(q)$:

$$r := \sinh^{-1}(Re(\sqrt{(q+1)}\,\overline{\sqrt{(q-1)}})) \quad \text{and}$$
$$s := 2\arctan(Im(\sqrt{(q-1)}) / Re(\sqrt{(q+1)})) .$$

Here the overscore signifies Complex Conjugate. Fortunately, recondite formulas like these have been found for all the elementary functions, and Dr. Joe Tanzini painstakingly microprogrammed them into the hp-15C.

Do not be misled by the foregoing illustrations into thinking either that complex variables are tricky, or that they will ever supplant finite elements. On the contrary, complex variables are as easy to use as real when implemented properly. And they supplement rather than supplant other numerical procedures. Experience with complex variables builds experience with conformal transformations that straighten corners, and with similar techniques that remove singularities analytically before they embarrass naive numerical methods. Helping students and teachers acquire and promulgate that experience is a part of the hp-15C's mission that I hope will soon be picked up by other computers, with bigger displays, capable of exhibiting conformal transformations graphically.

Display limitations appear also to inhibit matrix arithmetic on a calculator, but appearances are illusory. People rarely (perhaps too rarely) pay attention to values generated in intermediate calculations; and even when a displayed value is examined it serves at least as often to confirm that the correct variable has been accessed as to check whether its value is correct. Evidently a variable's name means more than its value. This observation led me to propose to Dennis Harms and Rich Carone that a calculator be built to display *Descriptors* instead of values for matrices. Whereas a calculator's scalar variables are named by their addresses, whereby we locate their values in memory, every matrix variable could be addressed by its name, each linked to a pointer to an otherwise anonymous array of values. This scheme requires dynamic memory management, which relieves the user of the hp-15C of any need to know where in memory reside his matrices (or the auxiliary stack for complex variables, or scratch space for the [SOLVE] and [∫ dx] keys.) The implementation of dynamic memory management and matrix input/output for the hp-15C fell to Eric Evett; Paul McClellan microcoded the matrix arithmetic operations. Details appear in [15] and [16], so an example here will suffice to show how easy they have made matrix computations.

Consider this 4x4 matrix A and its inverse:

$$A = \begin{vmatrix} 6 & -1 & -3 & 1 \\ -2 & 0 & 1 & 3 \\ 2 & -1 & 0 & 1 \\ -3 & 2 & -1 & 0 \end{vmatrix} ; \quad A^{-1} = \begin{vmatrix} -5 & -6 & 23 & 9 \\ -11 & -13 & 50 & 20 \\ -7 & -8 & 31 & 12 \\ -1 & -1 & 5 & 2 \end{vmatrix} .$$

These keystrokes enter A into the hp-15C :

```
4 ENTER DIM A          ... Declare that A is 4x4.
USER MATRIX 1          ... Initialize walk through matrix.
6    STO A   1 CHS STO A   3 CHS STO A   1   STO A
2 CHS STO A   0    STO A   1    STO A   3   STO A
2    STO A   1 CHS STO A   0    STO A   1   STO A
3 CHS STO A   2    STO A   1 CHS STO A   0   STO A
```

Each time [STO] [A] is pressed during this walk through the matrix **A** , "[A i,j]" displays momentarily to tell the user which element of which matrix is being altered. At the end of the walk, after "[A 4,4]" has been seen, all elements of **A** have received their values. This input takes about 40 sec.

The next few keystrokes compute **C** := A^{-1} :

RESULT C	... Tells hp-15C where to put A^{-1}.
RCL MATRIX A	... See [A 4 4] displayed.
[1/x]	... See [running] for 11 sec.,
	... then [C 4 4].

The displayed descriptor tells the user that a 4x4 matrix **C** resulted from the last operation and is now ready for the next. To view the 16 elements of **C** , press [RCL] [C] 16 times. Each time, "[C i,j]" will display for a moment, and then the value of $C_{i,j}$, where the indices i,j advance in lexicographic order from 1,1 to 4,4 . This walk takes about half a minute, or two minutes if the elements are copied onto paper, and shows

$$C = \begin{vmatrix} -5.000000049 & -6.000000059 & 23.00000022 & 9.000000085 \\ -11.00000011 & -13.00000013 & 50.00000048 & 20.00000019 \\ -7.000000067 & -8.000000080 & 31.00000030 & 12.00000012 \\ -1.000000011 & -1.000000013 & 5.000000048 & 2.000000019 \end{vmatrix}$$

A system of linear equations **Ad** = **b** can be solved for **d** = A^{-1}**b** without calculating A^{-1} . Instead, use the [÷] key thus; press
RESULT D RCL MATRIX B RCL MATRIX A [÷]
to display the descriptor of the solution **d** calculated faster and more accurately.

How accurate is **C** ? Were it not obvious, we would have to overestimate the loss of accuracy by computing a condition number $\|A^{-1}\|\,\|A\|$; the norm $\|...\|$ here can be any of three built into the hp-15C . The row-sum norm, Matrix Operation #7, is invoked thus:

RCL MATRIX C	MATRIX 7	... $\|A^{-1}\| \diamond \|C\| = 94.$
RCL MATRIX A	MATRIX 7	... $\|A\| = 11.$
[x]		... $\|C\|\,\|A\| = 1034.$

This indicates that somewhat less than 1034 ulps was lost to roundoff; the reasoning is explained in the chapters on matrix operations and errors in [15]. Also explained there is how to improve the accuracy of **d** by *Iterative Refinement*; the residual **c** = **b** - **Ad** is calculated in one step by Matrix Operation #6, and the solution **e** of **Ae** = **c** added to **d** . In this process, as in matrix multiplication and inversion, the hp-15C fares better than might be expected of a machine that carries ten sig. dec. For example, let **E** be a multiple of the notorious Hilbert matrix; $E_{i,j}$:= 360360/(i+j-1) for $1 \le i,j \le 8$. The constant 360360 ensures that every element of **E** is an integer, hence exact, and 8x8 is as large a matrix as fits in the calculator. In under 90 sec., it gets E^{-1} correct to roughly three sig. dec., three more than are expected in view of $\|E^{-1}\|\,\|E\| > 10^{10}$. This extra accuracy is no accident with ill-conditioned matrices like **E** , prone to systematic cancellation, but is due to extra-precise accumulation of scalar products to 13 sig. dec. during matrix operations.

The hp-15C does not refuse to invert a singular matrix **A** but instead inverts some nearby nearly indistinguishable **A**+Δ**A** ; since $\|(A+\Delta A)^{-1}\|$ must be huge, bigger than $1/\|\Delta A\|$, the nature of **A** is revealed. Because of this policy, one of the solutions **d** of a consistent system **Ad** = **b** will always be delivered with $\|d\|$ not much bigger than it has to be.

Least squares problems can be solved on the hp-15C by using the normal equations and simple programs, or by more robust programs based upon orthogonal factorization techniques like those in the book [18] by Lawson and Hanson, especially on pp. 66, 208-212, and 275. Programs of both kinds written by Paul McClellan appear in ch. 4 of [15] together with advice on when to use them. One of them can solve least squares problems with linear constraints and perform linear regression upon up to five independent variables with any arbitrarily large number of observations.

With machines like the hp-15C in their shirt-pockets, students of engineering, mathematics, science or statistics can practise what we preach in the first two years of college, ever more confident that what we teach will, as it should, serve them throughout their careers.

THE INTEL 18087 : Dr. John F. Palmer, a numerical analyst working for Intel in 1975, discerned the invidious possibility that two different computer systems inside one small box bearing the logo "Intel" might be unable to work upon numerical data in a shared memory for lack of a common format. He was asked to deal with this problem, and he asked me to help him design "the *very best* arithmetic" that could be implemented upon all the diverse microprocessors Intel was planning.

We chose binary formats with an implicit leading bit, very like I. Bennet Goldberg's variation [19], so the 32-bit *Single* and 64-bit *Double* formats have ranges and precisions usually better and never much worse than any formats available elsewhere in comparable wordsizes. An *Extended* format as wide as we dared (80 bits) was included to serve the same support role as the 13-decimal internal format serves in Hewlett-Packard's 10-decimal calculators (their 12-digit calculators use 15 digits). The tightest possible rounding, statistically unbiased, was specified for the arithmetic operations +, -, x, ÷, √ because we knew how and why. Finally, we provided ±∞ and a "Not-a-Number" symbol (*NaN*) because they are so valuable to those who have used them on the few computer architectures that include such things. They turn computer arithmetic into a system that is *formally closed*: every arithmetic operation, valid or not, now produces a result and also, whenever the operation is exceptional, a signal. The signal, called a *flag*, warns a program when a subprogram's result, if not obviously wrong, is questionable because an unpremeditated arithmetic exception may have occurred. Therefore, closure is no mere mathematical frill; now computation can proceed after an isolated invalid datum or a mistake rather than have to hang up and leave, say, the control surfaces of an aircraft stuck in an unusual position.

Our design was not so much new as eclectic; we chose the best that we could make work together in a system about which no user has to learn more than will matter to him.

Shortly after the design was announced [20] its single and double formats (but not its exception handling) appeared in a floating-point slave-processor chip, the Intel 8232, second-sourced

as the AMD 9512. Another implementation used up almost a third of the microcode in the Intel 432 microprocessor. So far, the most ambitious and most widely known implementation is Intel's 8087 coprocessor chip [21] that widens the instruction set of 8086 and 8088 microprocessors to include floating point arithmetic. Its features, listed in this paper's abstract and explained elsewhere [22 - 25], deserve only a few comments here.

Like Hewlett-Packard's elementary transcendental functions in its recent calculators, Intel's are accurate to within an ulp or two, but that ulp is in the 64th sig. bit, beyond 18 sig. dec. Both the calculators and the i8087 achieve their accuracies via digit-by-digit methods [26] that generate ln(1+x), exp(x)-1, tan(x) and arctan(x) quickly and correct to 64 sig. bits in the i8087, 13 sig. dec. in h-p calculators. Then simple but unobvious programs produce the other elementary functions accurately from those four. Intel's programs were written by Steve Baumel with my help, and appear in the CEL (Common Elementary function Library) in RMX-86 on the 86/330A. Their accuracies surpass crafty programs by Cody and Waite [27] run on less refined arithmetics.

Numerical programs that will run correctly on a computer after recompilation from some standard language like Fortran, or after some other almost mindless translation, are called *Importable* to that computer. The i8087 confers importability upon almost every program that runs upon several if not all diverse computer arithmetics. Indeed, experience [28] indicates that *Portable* programs, those designed to run universally, can be made simpler, shorter and faster when adapted to run on an i8087. For two years the main obstacle to its use has been a dearth of compilers that would generate code to exploit it in the many computers that have one, among them the IBM PC. Except Intel's, those early compilers that served the i8087 hedged against its possible unavailability by using only whatever subset of its capabilities could be emulated easily in software. Now that the chip is abundant such a policy is no longer economical, and language processors that use the chip efficiently are or soon will be available for APL, C, Forth, Fortran and Pascal, and with several operating systems. I have obtained good results from Intel's Fortran running in RMX-86 on an Intel 86/330A, and from APL*PLUS™/PC by STSC Inc. on an IBM PC ; and the latest versions of Fortran on the IBM PC are getting pretty good.

THE PROPOSED IEEE STANDARD: Mathematical craftsmanship can be shared as computer software designed to be used conveniently by people among whom most will understand its mathematics little better than most motorists understand their cars drive trains. But numerous obstacles impede the dissemination among computers of programs as easy to use as are the keys of calculators discussed above. One of those obstacles is gratuitous: computer arithmetics are too diverse and, as we have seen earlier in this paper, occasionally too capricious to allow programs so delicate as those in the calculators to be copied mindlessly onto other machines with no risk of malfunction. Portable programs demand craftsmanship of another kind, capable of coping with the vagaries of computers and compilers without sacrificing too much accuracy, speed or convenience. Among the monuments to that craftsmanship are the EISPACK [29], LINPACK [30] and PORT [31] libraries of Fortran codes. Some portable libraries are less satisfactory; the portable elementary functions coded in C in the UNIX™ system are slow and inaccurate, and tend to be replaced by programs (often unnecessarily in assembly language) that may be worse but ought to be at least as good as those in the book [27] by Cody and Waite if chosen properly for the machine. Commercially distributed libraries like IMSL's [32, 33] and NAG's [34] must forego a measure of portability to regain reliability and speed; these libraries are distributed in versions each tuned to the one computer and compiler on which it will run. They are not available for computer systems too unlike others and too little used to repay the cost of putting together and maintaining another version.

I would like to believe that these considerations weighed upon all our minds when, responding to Dr. Robert G. Stewart's invitation late in 1977, we convened to draft a floating-point arithmetic standard. Intel's plans to build the i8087 also influenced us, if only by lending credibility to the otherwise utopian "KCS proposal" advocated by myself, Jerome T. Coonen and Prof. Harold S. Stone, and derived from the Intel design by refinement and extension [35]. Implementations [36, 37], analyses [35, 38, 39] and especially amendments and simplifications by Coonen led in 1980 to the tentative adoption [40] of the KCS proposal, despite its unusual features, as the basis of a draft IEEE standard p754. Its most controversial feature was *Gradual Underflow*, a scheme implicit in Goldberg's variation [19] but exploited hitherto by almost nobody but me [41]. This scheme enforces a kind of closure property described precisely by insisting that the Theorem about $p - q$, cited above while discussing the area of triangles, be true without the clause " unless $p - q$ suffers exponent underflow. " Consequently, the calculated value of $p - q$ is zero just when $p = q$. More important, gradual underflow differs from the usual schemes because it *almost never* (but, alas, not *never*) generates more numerical uncertainty than roundoff does, so it enhances the provable [38-41] reliability of many equation-solving codes, among others. But it costs something to implement, so it remains controversial even if much ado about very little.

P754, like the i8087, is a closed arithmetic system that, by default, supplies a result and raises a flag for every exceptional arithmetic operation. The default results are these:

Exception Type	Default Result
Invalid Operation	NaN (Not a Number)
Overflow	$\pm\infty$ and signal Inexact too
Divide-by-Zero	$\pm\infty$ exactly
Underflow	Gradual → subnormal numbers
Inexact	Correctly rounded result

Of course, exceptions are by nature inimical to any single preselected default result. NaN may be the best single response to 0/0 or ∞/∞ or $\sqrt{(-3)}$; but APL programmers expect $0/0 = 1$, and others may prefer to stop on that occasion. P754 does allow the implementor, at his option, to provide *Traps* whereby a user may select such non-default responses to exceptions as he likes. Also, NaNs may be used for uninitialized and/or missing data, and for retrospective diagnostics. And the implementor is obliged to offer the user a choice of four rounding algorithms in case the default is unsatisfactory. A discussion of these features would burst beyond the space allowed for this paper, so a few final comments must suffice.

Despite a residuum of controversy and uncertainty about how higher-level languages will interface to its unusual features [35, 42-46], p754 has been adopted by surprisingly many manufacturers, with complete implementations ranging in speed from about a thousand floating-point operations per second in an Apple III [47] to three million in an ELXSI 6400 [48], with others like Intel, Hewlett-Packard [49], National Semiconductor, Motorola and Zilog [42] in between, to mention only the best known firms. However p754 is not an official standard; although its final draft (#10) was finished in Dec. 1982, it has not been endorsed yet by the IEEE , nor is it available yet from what must ultimately be its source:

IEEE, 345 E. 47th St., New York NY 10017. An earlier draft #8 [40] was no sooner published for public comment than it was adopted officially by the International Electrotechnical Commission in Geneva, but that is an inferior version, much harder to understand and to implement; don't use it. Draft #10 of p754 is available now from

Richard Karpinski, UCSF U-76,
San Francisco, Calif. 94143

Available from this same source is draft #1 of a proposal p854 [50] that generalizes p754 from binary arithmetic to decimal and allows other word-sizes than just 32 or 64 bits. Like a much earlier proposal [51], p854 looks forward to the day when the numbers humans see will be the numbers they deserve.

ACKNOWLEDGMENTS: I am grateful for support from the U. S. Office of Naval Research (N 00014-76-C-0013) and from the Dept. of Energy (DE-AMOS-76SF00034 & DE-AT03-79ER10358) for my work on the proposed standards. Thanks are due also to a few Berkeley graduate students and former students: R. P. Corbett, Dr. J. W. Demmel, Dr. D. Hough, G. S. Taylor and especially J. T. Coonen.

REFERENCES:

[1] D. B. DeLury "Computation with Approximate Numbers" The Mathematics Teacher 51 (1958) pp.521-530.

[2] W. Kahan "A Survey of Error Analysis" in "Info. Processing 71" (1972) pp.1214-1239; North-Holland Publ. Co., Amsterdam.

[3] W. Kahan & B. N. Parlett "Can You Count on your Calculator?" translated by W. Frangen into "Können Sie sich auf Ihren Rechner verlassen?" in "Jahrbuch überblicke Mathematik 1978" ss.199-216; Bibliographisches Institut AG , Mannheim.

[4] W. Kahan "Interval Options in the Proposed IEEE Floating Point Arithmetic Standard" in "Interval Mathematics" ed. by K. Nickel (1980) pp.99-128; Academic Press, New York.

[5] W. Kahan "Why do we need a floating-point arithmetic standard?" in preparation.

[6] The Appendix "Accuracy of Numerical Calculations", pp.96-211 in reference [15] below.

[7] R. E. Martin "Printing Financial Calculator Sets New Standards for Accuracy and Capability" Hewlett- Packard Journal 29 #2 (Oct. 1977) pp.22-28.

[8] D. W. Hanes "Improved Algorithms: Making $2^3 = 8$ " in Session 32 "Advanced Pocket Calculators" of the IEEE ELECTRO 76 in Boston, May 11-14 1976 . An extract appears in the Hewlett-Packard Journal 28 #3 (Nov.1976) pp.16-17 .

[9] W. Kahan "Personal Calculator Has Key to Solve Any Equation $f(x) = 0$ " Hewlett-Packard J'l 30 #12 (Dec. 1979) pp.20-26.

[10] W. Kahan "Handheld Calculator Evaluates Integrals" Hewlett-Packard J'l 31 #8 (Aug. 1981) pp.23-32.

[11] Mary H. Payne "Floating Point Basics and Techniques" in the proceedings of the Spring DECUS Symposium held in St. Louis May 23-27, 1983.

[12] M. H. Payne and R. W. Hanek "Radian Reduction for Trigonometric Functions" SIGNUM Newsletter 16 (Jan.1983) pp. 19-24, and a sequel in Newsletter 17 (Apr. 1983) pp.18-19 .

[13] H. G. Diamond "Stability of Rounded Off Inverses Under Iteration" Math. of Comp. 32 (1978) pp. 227-32.

[14] P. Henrici "Essentials of Numerical Analysis (with Pocket Calculator Demonstrations)", and its "Solutions Manual" (1982); Wiley & Sons, New York.

[15] "HP-15C Advanced Functions Handbook" (1982) part # 00015-90011; Hewlett-Packard, Corvallis, Oregon.

[16] E. A. Evett, P. J. McClellan and J. P. Tanzini "Scientific Pocket Calculator Extends Range of Built- in Functions" Hewlett-Packard Journal 34 #5 (May 1983) pp.25-35.

[17] W. Kahan "Branch Cuts for Complex Elementary Functions" Report #PAM-105 (October, 1982) of the Center for Pure & Applied Mathematics, Univ. of Calif., Berkeley.

[18] Charles L. Lawson and Richard J. Hanson "Solving Least Squares Problems" (1974); Prentice-Hall, Englewood Cliffs, N. J.

[19] I. B. Goldberg "27 Bits are Not Enough for 8-Digit Accuracy" Comm. ACM 10 (1967) pp.105-108.

[20] J. Palmer "The Intel Standard for Floating-Point Arithmetic" Proc. IEEE COMPSAC 1977 pp.107-112.

[21] R. Navé "A Numeric Data Processor" Proc. IEEE Int'l Solid State Circuits Conference 1980 pp.108-109.

[22] "iAPX 86/20, 88,20 Numerics Supplement" in "Intel iAPX 86, 88 User's Manual" (1981); Intel, Santa Clara, Calif. 95051.

[23] R. Startz "8087 Applications and Programming for the IBM PC and Other PCs" (1983); Rob't J. Brady Co., Bowie, MD 20715.

[24] J. F. Palmer and S. P. Morse "The 8087 Primer" (1984); Wiley & Sons, New York.

[25] Tim Field "The IBM PC and the Intel 8087 Coprocessor" parts 1 and 2 in BYTE 8 (1983) Aug. pp.331-374 and Sept. pp.331-355.

[26] Part VI "Elementary Functions" of "Computer Arithmetic" ed. by E. E. Swartzlander Jr. (1980), v.21 of "Benchmark Papers in Elect. Eng. and Comp. Sci."; Academic Press, New York.

[27] W. J. Cody Jr. and W. Waite "Software Manual for the Elementary Functions" (1981); Prentice-Hall, Englewood Cliffs, N. J.

[28] Virginia Klema "Statistical Computations with IEEE Floating Point Arithmetic" ASA Stat. Comp. Sec. of Proc. 1983 Joint Statistical Meetings in Toronto, Canada, Aug. 16, 1983.

[29] B. T. Smith et al. "Matrix Eigensystem Routines - EISPACK Guide" 2d ed., Lecture Notes in Comp. Sci. vol. 6 (1976), and B. S. Garbow et al. "Matrix Eigensystem Routines - EISPACK Guide Extension" Lecture Notes ... vol. 51 (1977); Springer-Verlag, New York. The programs can be obtained on tape from either IMSL (see [32]) or the National Energy Software Center at Argonne National Labs., Argonne, Illinois 60439.

[30] J. J. Dongarra et al. "LINPACK Users' Guide" (1979); Society for Industrial and Applied Math., Philadelphia.

[31] Phyllis Fox, ed. "The PORT Mathematical Subroutine Library"; Computing Information Service, Bell Labs, Murray Hill, N. J.

[32] International Mathematical and Statistical Libraries (IMSL, Inc.) Houston, Texas 77036-5085, distributes Fortran libraries for a wide range of numerical and statistical computations.

[33] John R. Rice "Numerical Methods, Software and Analysis: IMSL" Reference Edition" (1983); McGraw-Hill, New York.

[34] The Numerical Algorithms Group (NAG, Ltd.), Oxford OX26NN, England, distributes ALGOL 68 and FORTRAN libraries of superb numerical subroutines.

[35] ACM SIGNUM Newsletter Special Issue on the Proposed IEEE Floating Point Standard, October 1979.

[36] J. T. Coonen "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic" COMPUTER 13 #1 (Jan. 1980) pp.68-79 and corrections thereto on p.62 of [40].

[37] G. S. Taylor "Compatible Hardware for Division and Square Root" pp.127-134 of Proc. 5th IEEE Symposium on Computer Arithmetic, Ann Arbor, Mich., May 1981. See also pp.190-196.

[38] S. Linnainmaa "Combatting the Effects of Underflow and Overflow in Determining Real Roots of Polynomials" ACM SIGNUM Newsletter 16 #2 (June 1981) pp.11-16.

[39] J. W. Demmel "Effects of Underflow on Solving Linear Systems" to appear in SIAM J. Sci. Stat. Comp.

[40] COMPUTER 14 #3 (Mar. 1981) pp.51-86 contain draft 8.0 of the proposed IEEE standard p754 for binary floating-point arithmetic, plus several articles that discuss it.

[41] W. Kahan "7094-II System Support for Numerical Analysis" SHARE Secretarial Distribution SSD-159, Item C4537 (1966).

[42] Session 16, "The New Floating-Point Standard: Implementation and Applications" in Proc. Mini/Micro West 1983 Computer Conference and Exhibition in San Francisco, Nov. 10, 1983.

[43] R. J. Fateman "High-Level Language Implications of the Proposed IEEE Floating-Point Standard" ACM Trans. Progg. Languages. and Systems. 4 (1982) pp.239-257.

[44] W. Kahan and J. T. Coonen "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments" pp.103-113 of "The Relationship between Numerical Numerical Computation and Programming Languages" ed. by J. K. Reid (1982); North Holland Publ. Co., Amsterdam.

[45] W. Kahan, J. Demmel and J. T. Coonen "Proposed Floating-Point Environmental Inquiries in Fortran" IEEE Floating-Point Subcommittee Working Document p754/82-2.17 (1982).

[46] R. P. Corbett "Enhanced Arithmetic for Fortran" ACM SIGNUM Newsletter 18 #1 (Jan. 1983) pp.24-28.

[47] "Numerics Manual: A Guide to Using the Apple III Pascal SANE and ELEMS Units" (1983) item 030-0660-A; Apple, Cupertino, CA 95014

[48] G. S. Taylor "Arithmetic on the ELXSI 6400" pp.110-115 of Proc. 6th IEEE Symp. on Comp. Arith., Aarhus, Denmark, June 1983.

[49] J. G. Fiasconaro "Instruction Set for a Single-Chip 32-Bit Processor" Hewlett-Packard Journal 34 #8 (Aug. 1983) pp.9-10.

[50] W. J. Cody Jr. "A Generalization of the Proposed IEEE Standard for Floating-Point Arithmetic" Proc. 15th Symposium on the Interface: Comp. Sci. and Statistics, Houston TX, Mar. 17, 1983

[51] F. N. Ris "A Unified Decimal Floating-Point Architecture for the Support of High-Level Languages (Extended Abstract)" ACM SIGNUM Newsletter 11 #3 (Oct. 1976) pp.18-23.

Mathematics Dept., and Elect. Eng. & Computer Science Dept.
University of California, Berkeley, California 94720.
Nov. 22, 1983.