HOW CRAY'S ARITHMETIC HURTS SCIENTIFIC COMPUTATION
and what might be done about it

W. Kahan
Elect. Eng. & Computer Science Dept.
University of California at Berkeley

ABSTRACT
CRAY's floating-point hardware runs fast but breaks mathematical rules honored nowadays by practically all computer arithmetics of commercial significance in scientific and engineering computation. Consequently valuable numerical software, provably infallible on most other machines, occasionally fails mysteriously on CRAYs. As speeds and memories grow, so does the risk of failure.

CRAY users suffer because they cannot merely recompile ostensibly portable software that works on workstations. To mitigate risks, software must instead be scrutinized for obscure hazards on CRAYs and encumbered in complications that compromise performance. For instance, CRAY's software DOUBLE PRECISION (REAL*16) runs about three times slower than it could if CRAY's floating-point add/subtract possessed the guard digit enjoyed by other computer arithmetics. Retrofitting a guard bit, as proposed herein, will not affect existing software on CRAYs adversely nor will it slow CRAY's cycle time.

Those of us who do not own a CRAY suffer from its arithmetic in a different way. We are denied software that would work well on our machines but, because it might fail on CRAYs, is ineligible for the Federal subsidies granted to research projects devoted, among other things, to the development of software portable to *all* supercomputers and especially CRAYs. For instance, the LAPACK project has had to eschew algorithms that would run better than all others on workstations but slowly or wrongly on CRAYs.

Ultimately CRAYs and workstations will approach compatibility with each other. Since almost all workstations conform to IEEE standard 754 for floating-point arithmetic, CRAY will have to do the same unless that standard falls by the wayside for lack of numerical software that exploits its superior features unavailable on CRAYs. In other companies some hardware designers, unaware of nasty implications for software, have come to regard CRAY's arithmetic as a licence to do no better. Already new designs have begun to appear that "support" certain IEEE standard features (some of them latent in CRAY hardware but unsupported by its software) only if new hardware is run in a slow mode unlikely to be mentioned in advertisements of benchmark performance. (The i860 is like that.) Therefore, all of us have a stake in the quality of CRAY equipment and software regardless of whether we use them, since their example influences other producers.

Prepared for the CRAY User Group meeting in Toronto, Canada, Apr. 10, 1990

Table of Contents

ABSTRACT 1
Table of Contents 2

INTRODUCTION 3
Table 1: Maximum Dispacement ... 5
Table 2: A COMMUNITY THAT EXCLUDES CRAYS 5

WHAT ARE THE CHANCES THAT A NUMERICAL PROGRAM WILL FAIL? 6
Ill-Condition ≐ Near-Singularity 6
Estimating the Probability of Failure 7
Risk Equation 8
Predictions of Risk 8
Table 3: Significant Bits for Fortran REAL*8 9

CRAY'S ADDITION AND SUBTRACTION 10
Where is CRAY's Missing Guard Bit? 10
AREA of a Triangle ..., Program RATAREA 11
Table 4: Results from Program RATAREA 11
A Spurious Backward Error-Analysis 13
Figure 1: Noise Leaks into the Feedback Path 14
Benign Singularities, L(y), V(N, i) 14
Confused Comparisons on the CRAY 2 16
NEXT1 Programmed Transportably ... 18
Divided Differences 19
The Pessimism of *Almost*^{2} *Right* Error Analysis on a CRAY 21
Programs to Compute A(z) and arccosh 23
Figure 2: A(z) 23
Compensated Summation 24
Figure 3: Error in Trajectory Calculations 25
CMPSUM: a Program to Sum a Long Series 27
Table 5: Results from Program CMPSUM 28
Software-Simulated Double-Precision Retarded 28

CRAY'S MULTIPLICATION AND DIVISION 29
AMOD Misbehaves 30
Sethian gets into Trouble 30
Monotonicity Fails 31
A Quadratic Equation becomes Unreal 31
Simple Specifications rendered Complicated 31
Paranoid Programmers 32
Accuracy (Eps) vs. *Precision* 31
The Importance of the Relative Error Bound Eps 34
Stopping Criteria Confounded 34
An Apology to Language Implementors 35

ANOTHER PROGRAM WE CAN'T HAVE 36

MUST EXCEPTIONS BE FATAL ERRORS ? 38

RECOMMENDATIONS FOR CHANGE 40

ACKNOWLEDGMENTS 42

This is WORK IN PROGRESS ! It is a draft circulating to elicit helpful comments. It is not yet ready for promulgation. W. K.

INTRODUCTION

To advocate a change in CRAY's arithmetic is, I know, to swim against the current. Many who read this take justifiable pride in the speed with which CRAYs carry out computations indispensable to progress in Science, Technology and National Security. If CRAY's arithmetic is so sinful that it must be changed, and if " the wages of sin is death " (*Romans* 6:23), why are so many of you still walking about? You must have triumphed over aberrations built into CRAY's arithmetic; of that triumph too you can be justly proud, but what did it cost you?

How much time has been spent to put into numerical software those extra precautions occasioned only by the idiosyncracies of CRAY's floating-point arithmetic? How much do they penalize performance?

Only recently have some major changes in the computing ambience rendered more nearly obvious the cost of coping with perversely designed arithmetic. One change has been the disappearance of a few perverse arithmetics, notably those descended from the CDC 6600 and the Univac 1107, leaving CRAY standing tall like a tree amidst a forest of stumps. Many a super-computing mainframe has been felled by its disadvantageous price/performance compared with powerful new workstations. Workstations are replacing the terminals through which users used to access their CRAYs. Now a user typically has access to a network of workstations and other computers among which may be several CRAYs. Consequently a user whose only computer used to be his CRAY can now obtain results from diverse machines, and compare them. Results differ.

Results differ between CRAYs and other machines, and between one CRAY and another; and when the difference is too big to ignore a CRAY is too often in the wrong. *Invidious Comparison* is the new phenomenon that now disturbs many a CRAY user who, ten years ago, would have believed along with CRAY's designers that variations originating in the 48th sig. bit can neither rise up so often nor smite so hard. That belief was mistaken.

For instance, the results displayed in Table 1 are taken from a computation making the rounds by electronic mail. O. O. Storaasli of NASA Langley Research Center, Hampton VA 23665-5225, sent them to me. The computation is a Finite-Element analysis with 16146 simultaneous equations. That the 53-sig.-bit machines are about 32 times as accurate as the 48-sig.-bit CRAY 2 comes as no surprise, but why the 48-sig.-bit CRAY Y-MP should be 32 to 111 times worse than the CRAY 2 is unclear; can the chopped add/subtract on the Y-MP be that much inferior to the rounded arithmetics on the other machines? Questions like this about a computation that costs over 10^9 floating-point operations will not be answered by comparing results on a hand-held calculator.

The examples in this paper are designed to be debugged by hand and yet not easily. They are designed to reveal how CRAYs can get plausible but wrong results from ostensibly portable software that executes provably reliably on all other computers now commercially significant for scientific and engineering computation. Table 2 lists some of those other computers. I do not allege that they can compute something a CRAY cannot. Rather, they form a large community from which CRAY is excluded, a community that shares

numerical algorithms and software written in a transparent natural style. Those algorithms will work on a CRAY only after they are modified to compensate for idiosyncracies in CRAY's floating-point arithmetic. The modifications encumber algorithms with *fudge-factors*, tests and branches unnecessary in the community at large, sometimes transforming a simple fast algorithm into a slow complicated one, sometimes jeopardizing accuracy too.

But very slight changes to CRAY's arithmetic would obviate those complications, with dramatic effect upon numerical software:

Almost all ostensibly portable numerical software, designed to run after recompilation indiscriminately on all computers made by IBM and DEC and SUN and most others, would run also on CRAYs with results only slightly (6 sig. bits) less accurate.

That is the motive for this paper: to change CRAY's arithmetic. I have wished to do so for a decade. And now I am not alone.

David H. Bailey *et al.*, in "Floating Point Arithmetic in Future Supercomputers" *Int'l J. of Supercomputer Applications* **3** (Fall 1989) 86-90, advocated that supercomputers adopt IEEE standard 754 for binary floating-point arithmetic although it was intended at first for microcomputers. (For a description of that standard see W. J. Cody *et al.* "A Proposed ... Floating-point Arithmetic" IEEE *MICRO*, Aug. 1984, 86-100.) I think they are right.

In the long run CRAY Research Inc. will have to bite the bullet and switch to IEEE 754. What will CRI do in the short run?

Since IEEE 754 is notoriously harder than CRAY's present arithmetic to build correctly and fast, CRAY engineers will need time to do the job right. And something has to be done for users of CRAYs currently installed. Compared with conversion to IEEE 754, my interim proposal herein is extremely modest:

- It retains CRAY's present floating-point format, leaving all files of binary data unaffected.
- It does not require recompilation of existing Fortran codes.
- It will not noticeably slow CRAY's arithmetic.
- I think it can retrofit advantageously onto existing CRAYs.

I propose that CRAY Research Inc. retrofit all CRAYs with one guard bit in Subtraction. Other improvements are less important.

As this is written, CRAY engineers are mulling over my proposal and their other options. I am grateful to several of them for trying to help me get the facts straight in this document, but they too are mystified by parts of CRAY's manuals. For lack of officially authorised access to the detailed specifications and simulators that tell exactly what different CRAYs do, I have had to speculate like an astronomer, drawing possibly incorrect inferences from remote observations; I request your indulgence.

Table 1 : Maximum Displacement *computed by different arithmetics*

Computer Arithmetic	Carried No. Sig. Bits	Computed Result
CRAY 2 Double Precision	96	0.447440 341220 910663
Convex 220 REAL*8 Non-IEEE 754	53	0.447440 33*9995 944*
Silicon Graphics Iris IEEE 754	53	0.447440 33*9938 063*
CRAY 2 REAL*8 ASCII data	48	0.447440 3*03*
CRAY 2 REAL*8 Binary data	48	0.447440 *472*
CRAY Y-MP REAL*8 ASCII data	48	0.447436 *106569*
CRAY Y-MP REAL*8 Binary data	48	0.447436 *255079 9*

Table 2 : **A COMMUNITY THAT EXCLUDES CRAYS :**
Computers Sharing Numerical Algorithms
Despite Differences in Arithmetic Hardware

Hexadecimal Floating-point, 4- and 8-byte words:
- IBM '370 series, including 3090.
- Clones by Amdahl, Fujitsu, Hitachi, Siemens, ...
- Near-clones by Data General, ...

Octal Floating-point, 52-bit words:
- Burroughs B6700's descendants, if any still exist.

Binary Floating-point, 36- and 72-bit words:
- DEC 10 and 20, if any still exist.

Binary Floating-point, 4- and 8-byte words, non-IEEE 754 :
- DEC PDP-11 and VAX, including VAX-vector 6000.
- Hewlett-Packard 3000s, if any still exist.

IEEE 754 Binary Floating-point, 4- and 8-byte words:
- IBM RS-6000.
- Sun 3, 4 and Sparc, and clones by Solborne.
- MIPS 2000, 3000, 6000. DECstation 3100.
- Silicon Graphics Iris. Stardent.
- Hewlett-Packard. Apollo.
- NCUBE. Inmos Transputers.
- Machines using chips by
  - National Semi., BIT, Weitek, TI, AMD, IIT, ...
- ...

IEEE 754 Binary Floating-point, 4-, 8-byte, and wider words:
- IBM PC and PS/2 descendants and innumerable clones.
- AT&T. Sun 386-based.
- Apple 2 and Macintosh. NEXT.
- Machines using chips by
  - AT&T, Motorola, Cyrix, Intel 80x86/80x87, 80960.

Decimal Floating-point:
- Hewlett-Packard Programmable Calculators built after 1974.

WHAT ARE THE CHANCES THAT A NUMERICAL PROGRAM WILL FAIL?
I do not allege that many aircraft will crash, bridges collapse and buildings crumble because of software failures on CRAYs. No conscientious designer trusts the results of one computation; he will insist upon corroboration from different methods and varied data before risking human life or large capital expenditures. If numerical errors cause harm they are usually part of a simulation of a proposed design which the computer declares unfeasible, so nothing will be done to expose the errors until after a competitor achieves what now seems impossible. This is the worst price we pay for unreliable software; second worst is the time spent on attempts to correct software we distrust. Third by far is the price paid for calamities caused by computers' numerical errors.

Generalities about the probabilities of software failure cannot be quantitative enough that actuaries could estimate the premiums for insurance against numerical calamities. The probabilities of rare failures are uncertain by orders of magnitude except that they are small. The consequences of a calamity are uncertain by orders of magnitude except that they may be enormous. Premiums must depend partly upon estimates of the average cost of calamities, a sum of products like (*probability of calamity*)*(*consequential cost*), which is unknowable. Nevertheless something worth knowing can be said about how the probabilities of software failure depend upon such factors as the level of roundoff and the dimensions of data.

Ill-Condition, which indicates that results are hypersensitive to tiny variations in data, has probabilistic aspects that have been the subject of publications by researchers ranging from John von Neumann in the late 1940s to Jim Demmel, Alan Edelman and Steve Smale, among others, in the last few years. The following explanation brutally condenses their work and my own.

Ill-Condition ≐ Near-Singularity
A problem that is excessively ill-conditioned for a given computer is probably insoluble on that machine without recourse either to software-simulated higher precision for both data and intermediate calculations, or else to some reformulation of the problem that involves possibly devious analysis and certainly exact symbolic manipulation. Attempts to solve the problem without resorting to either expedient usually fail. Other failure modes exist too.

What causes a numerical program to fail is approaching too near a *singularity* or passing beyond it. A singularity is a boundary point of the domain within which everything being computed is an analytic function of all its data and intermediate variables. A problem is regarded as *numerically unstable*, too *ill conditioned*, just when its data is too close to a singularity, regardless of algorithms chosen to solve the problem. An algorithm and also its program is regarded as numerically unstable just when some of the singularities are crossed by variations in intermediate results, not by variations in the given data. In other words, we do not blame a program for failing to solve accurately a problem that is intrinsically intractable because its solution is hypersensitive to unavoidable variations in its data. We do condemn a program when it produces poor results for data it seems to dislike even though the problem's solution is unexceptionable for that data.

For a fixed set of data, a program that seems unstable on one machine can be stable on a second that carries sufficiently more figures because roundoff on the latter is negligible compared with the distance to any nearby singularity, whereas roundoff on the former can carry the computation across that singularity. That is why Double Precision can succeed where Single Precision fails.

There is another way, less well appreciated, in which a program can be unstable on one machine and yet stable on a second; the first may be capable of errors or variations of a kind that cannot happen on the second. CRAYs suffer from this failure mode; they encounter singularities where other machines don't, as we'll see.

The closer data comes to a singularity, the worse will be the accuracy of computed results. In most cases the number of correct figures in results roughly equals the logarithm of the distance, from the data to the nearest singularity, measured in units of roundoff. That is why carrying another significant digit during computation usually produces another correct significant digit in results.

(The weasel-words "usually" and "In most cases" are made necessary by cases in which only a fraction of additional figures carried turn up in final results. For instance, when computing unwittingly what turns out to be a double root, we get only about half as many correct figures as we carry. In such a case a square root of a negative number is more likely than division by zero to expose this singularity, which is rarer than most other kinds. Another kind of singularity, more like an exponential at infinity than a pole, is associated with excessively large step-sizes that violate stability criteria for differential equation solvers; if the step-size is regarded as part rather of the program than the input data, the present discussion remains applicable.)

Estimating the Probability of Failure

Inadequate accuracy or a rude message is the result when numerical software fails, and evidently this happens when data comes closer than a roughly specifiable distance to a singularity germane to that software. Therefore we can estimate the probability of such failure by calculating that fraction, of the volume occupied by all allowed data, that is closer than a specifiable distance to a germane singularity. Such a calculation becomes feasible when we know enough about the locus of germane singularities.

The singularities studied in the literature are almost exclusively those germane to a problem rather than a program. The distinction matters because many a program fails at data for which the problem has an acceptable solution that the program cannot find accurately or at all. For instance, consider a program that solves systems of linear equations without pivotal exchanges. The singularity germane to a linear system is a singular matrix, which occurs at data where its determinant vanishes. The additional singularities germane to the program occur at data where certain subdeterminants vanish, corresponding to vanishing pivots; knowledgeable users will restrict this program to data (positive definite systems or systems with dominant diagonals, etc.) farther from additional singularities peculiar to the program than from the singularity intrinsic to the problem.

Whether intrinsic to the problem being solved or artifacts of the program, most singularities have the same degrading effect upon the accuracy of results. There are a few exceptions, mainly singularities that affect only the iterates in a self-correcting iteration; these singularities are more likely to stop iteration or prolong it than degrade final accuracy. We shall not ignore these exceptional cases, but they detract little from our final conclusion, which can be summarized in one *Risk Equation* thus:

Risk Equation: $DP \doteq C M^K S r$.

Here DP is the daily probability of failure, the expected rate of software failures per day. M is the amount of memory in use; larger memories go with larger data sets, higher dimensions and longer calculations. S is the speed of the arithmetic; faster machines are exposed to risk more often. The roundoff level r indicates the accuracy of the floating-point hardware; smaller error entails lower risk. C and K are positive constants that depend upon the kind of software in use and the quality of data, and also upon the quality of arithmetic (whether it is rounded or chopped, etc.); and usually $K > 1$.

Predictions of Risk

Were records kept correlating software failures with the amounts of memory and the kinds of software in use, empirical estimates of the constants C and K could be computed and used in the Risk Equation to predict expected rates of numerical software failure among sufficiently similar machines. The machines in Table 2 are similar enough that I expect the constants C and K to vary relatively little among them. (The IBM '370 family's chopped arithmetic is slightly riskier than others' rounded arithmetics in so far as chopped arithmetic suffers more from bias. And risks from exponent over/underflow have been neglected though they are noticeable in the D-formats on DEC PDP-11s and VAXes.)

The Risk Equation predicts failure rates DP generally worse for CRAYs than for most machines in Table 2. The excess is due in part to singularities that CRAYs can encounter where other machines cannot, as will become evident later. Most of these extra singularities can be removed by rewriting software to take CRAYs' idiosyncratic arithmetics into account without sacrificing portability to the machines in Table 2, and then the rewritten software will have nearly the same values C and K for CRAYs as for other machines. Even so, CRAYs end up with higher failure rates for reasons that can be inferred partly from Table 3.

Table 3 shows how accurately some of CRAY's competitors perform their floating-point arithmetic upon 8-byte words. The accuracy figures take into account both the numbers of significant bits carried and how the last bits are rounded off. For the IBM 3090, "chopped" arithmetic loses one bit and the hexadecimal format can lose up to three more. For CRAYs, the figures vary because the CRAY 2 "rounds" add/subtract differently than other CRAYs "chop" them, and regardless of whether multiply, divide and square root are "rounded" or "chopped" they err in diverse ways I have yet to figure out fully from CRAY's manuals.

Table 3 : Significant Bits for Fortran REAL*8 Arithmetic

Computer Family	Radix	Precision	vs.	Accuracy
CRAY 1, X-MP, Y-MP, 2	2	48		45 to 46
IBM '370, ... 3090	16	53 to 56		52 to 55
IEEE 754 Double Precision	2	53		53
DEC VAX G format	2	53		53
D format	2	56		56

Rounding error level: $r = 2^{-(\text{No. of sig. bits of Accuracy})}$

Table 3 shows that CRAY's roundoff level $r = 2^{-45}$ to 2^{-46} exceeds its competitors' by at least an order of magnitude or two. CRAY's memory sizes M are bigger too by an order of magnitude or two. And CRAY's speed S is also bigger by about an order of magnitude. Therefore, the Risk Equation predicts software failure rates bigger for CRAYs than for its competitors by some orders of magnitude. Since the bigger memory sizes have come into use among CRAYs only recently, CRAY users might expect to see increasing failure rates. But many of these failures will provoke programmers into using Double Precision to diminish r despite its terrible performance penalty; therefore casual onlookers may observe not an increasing failure rate but rather an increasing incidence of Double Precision, which is less noticeable.

The risk that CRAY Research Inc. cannot ignore is the possible appearance on the market of a supercomputer with roughly CRAY's speed, roughly CRAY's memory, arithmetic conforming to IEEE standard 754, and a significantly lower software failure rate.

If my Risk Equation is correct, CRAYs must now suffer severely higher failure rates than do other machines for similar numerical software. Unfortunately, the computing industry does not collect statistics that would confirm or contradict this prediction. That is why I must offer a host of examples and their analyses to show how failures can happen. If the examples seem a little contrived, yet are they far easier to understand than the puzzles encountered in real life.

CRAY's ADDITION AND SUBTRACTION

Most software failures traceable to CRAY's anomalous arithmetic are caused by the peculiar ways CRAYs subtract. For instance, the program AREA presented nearby realizes a numerically stable version of Heron's classical formula for the area of a triangle with given side-lengths. Regardless of the triangle's shape, the program is guaranteed accurate (in the absence of over/underflow) on any computer listed in Table 2 ; a proof of accuracy despite roundoff may be adapted from ex. 23, p. 152, of *Floating-Point Computation* by Pat H. Sterbenz (1974), Prentice-Hall, New Jersey. The proof requires this property of computer arithmetic:
If $1/2 \le A/B \le 2$ *then* A - B *is computed exactly* (unless it underflows).
Lacking this property, CRAYs lose AREA's accuracy as the shape of the triangle approaches a needle. All accuracy evaporates in extreme cases like those generated by a little program RATAREA shown above its results in Table 4 . Only CRAYs dislike it.

It may be hard to believe that a program so short and innocuous-looking as AREA can fail on a CRAY and yet work correctly on practically everything else. What goes wrong? The two operations that cause trouble are the subtractions A-B and C-D underscored ### in the program's text. Were they replaced respectively by

SNGL(DBLE(A) - B) and SNGL(DBLE(C) - D)

in Fortran on CRAYs, and only on CRAYs, the program would get provably correct results on CRAYs too; but this expedient is too uneconomical to be recommended as a general practice. It serves here merely to confirm our diagnosis.

What goes wrong with AREA goes wrong similarly with a host of other programs when they are run on CRAYs ; examples include areas of spherical as well as plane triangles, angles, side-lengths, radii of circumscribed and inscribed circles, and other trigonometry. Different kinds of examples will be examined later.

Where is CRAY's Missing Guard Bit ?

The CRAY X-MP and Y-MP subtract differently than the CRAY 2 , but similarly enough that most of the description can be shared.

Steps in CRAY's Subtraction D := A - B :
~~~~~~

- ▶1: Swap if necessary to get $|A| \ge |B|$.
- ▶2: Shift B's sig. bits right enough to equalize exponents.
- ▶3: On a CRAY 2, round off B's sig. bits past A's 48th.
  On a CRAY ?-MP, discard B's sig. bits past A's 48th.
- ▶4: Subtract what is left of B from A .
- ▶5: Shift left or right to normalize the difference.
- ▶6: Discard any bit past the result's 48th.

If the foregoing does not match exactly what CRAY's manuals say, yet I think it matches what they intended to say. "Discard" here means to chop bits off the magnitude, possibly diminishing it. "Round" means to add 1 to the magnitude's bit just right of the last to be retained, and then chop. Rounding seems to imply an extra carry propagation, but the hardware performs any rounding increment in ▶3 simultaneously with the subtraction in ▶4 . Addition A + B is just A - (-B) .
~~~~~~

AREA of a Triangle given its Side-Lengths
Programmed Ostensibly Portably in a Nondescript Language

```
Real Function  AREA( Real Values  A, B, C ) :
    ...  := area of a triangle with given side-lengths  A, B, C.
    ...  Note that arguments are copies passed by  Value.
    ...  First sort the arguments:
    If  A < B  then  SWAP(A, B) ;
    If  B < C  then  {  SWAP(B, C) ;
                        if  A < B  then  SWAP(A, B)  } ;
    ...  Now  A ≥ B ≥ C .  Next,  test for valid data:
    D := A-B ;
    ...  ### inaccurate only on a  CRAY.
    If D > C then  {
        Protest( "AREA(A, B, C)  has arguments  A =", A, " ," );
        Protest( " B =", B, "  and  C =", C );
        Protest( "that aren't a triangle's sides." );  Stop } ;
    Return  AREA := SQRT( (C+B+A)*(C-D)*(C+D)*((B-C)+A) )/4 ;
    End AREA .                 ...  ### inaccurate only on a  CRAY.


Program  RATAREA :
    ...  displays ratios  R and S  of areas of four needle-shaped
    ...  triangles for which only  CRAYs  get incorrect results
    ...  that disagree with  RX and SX ,  as shown in  Table 4 .
    ...  First create test data Y and Z with odd least-sig. digs.:
    Y := NEXT1(2.0) ;  ...  =  1.0 + (anything tiny enough).
    H := 0.5 ;           ... ( These data Y and Z are not critical.)
    Z := NEXT1(H) ;      ...  =  1.0 - (anything else tiny enough).
    T := 2.0*((H-Z) + H) ;
    ...  Next predict approXimately what the ratios ought to be:
    RX := SQRT(1.0/(1.0+Y)) ;  SX := SQRT(Z*0.75) ;
    ...  Finally compute ratios of areas of special triangles:
    R := AREA(2.0, Y, 1.0)/AREA(2.0, Y, Y) ;
    S := AREA(1.0, Z, T)/AREA(1.0, 1.0, T) ;
    Display  R, RX ; ...  they should agree near   0.7071... .
    Display  S, SX ; ...  they should agree near   0.866...  .
    End RATAREA .
```

Table 4 : Results from Program RATAREA on Various Machines

Computer Family	R	S
CRAY 1, X-MP, Y-MP	0.*0*	0.*0*
CRAY Double Precision (All)	0.*0*	0.*0*
CRAY 2	0.*81649 658*...	0.*99999 999*...
All machines in Table 2	0.70710 678...	0.86602 540...
Correct results (RX, SX)	0.70710 678...	0.86602 540...

Addition and subtraction suffer less from statistical bias on a CRAY 2 than on a CRAY ?-MP provided operands have sufficiently different magnitudes. That bias matters enough to account for the discrepancies Storaasli reported; see Table 1. His results were obtained by solving a huge system of linear equations with a positive definite matrix; the bias on the Y-MP had the same effect as increasing each diagonal element of that matrix by about 100 units in its last place while altering the relatively smaller off-diagonal elements in ways less predictable and less important. Increasing the diagonal elements of a positive definite matrix diminishes its inverse and tends to diminish the biggest elements of the equations' solution to an extent comparable with what we see in Table 1 . Whether the foregoing explanation accounts for the discrepancies fully remains to be seen; since I have looked at neither Storaasli's program nor his data, my diagnosis is no better than a diagnosis of cancer carried out over the telephone.

When A and B come relatively close the qualities of differences A - B computed on diverse CRAYs diverge from one another and from almost all other machines in ways too drastic to be hidden in statistics. What happens will be described here as if arithmetic carried only 8 sig. bits instead of 48. Suppose A = 8.0 and B is the 8 sig. bit floating-point number next smaller than A , namely B = 7.96875 . Written in binary, they are A = 1000.0000 and B = 111.11111 . Let's try to compute D := A - B :

	Like other machines	Like a CRAY ?-MP	Like a CRAY 2
A	1000.0000	1000.0000	1000.0000
-B	-0111.11111	-0111.1111*1*	-0111.1111*1*
chop/round		+*1*	-*1*
D	0000.00001	0000.000*10*	0000.000*00*

Other machines carry internally an extra *guard digit* that makes their difference D = A - B exactly. The CRAY ?-MP lacks that guard bit, so it drops B's last bit; this has the same effect as adding 1 in that position to cancel out a last bit that would otherwise have been subtracted from A . The CRAY 2 lacks the guard bit too; it injects B's last bit into the subtraction in A's last bit position instead of B's last, which is tantamount to subtracting that last bit twice. Consequently, the CRAY ?-MP gets a difference D twice as big as it should be, and the CRAY 2 gets D = 0 . (That zero is ominous; later we shall see why.)

CRAYs are not the first machines to compute differences blighted by lack of a guard digit. The earliest IBM '360s, from 1964 to 1967, subtracted and multiplied without a hexadecimal guard digit until SHARE, the IBM mainframe user group, discovered why the consequential anomalies were intolerable and so compelled a guard digit to be retrofitted. The earliest Hewlett-Packard financial calculator, the HP-80, had a similar problem. Even now, many a calculator (but not Hewlett-Packard's) lacks a guard digit. And a pair of floating-point coprocessor chips, the IIT 2C87 and 3C87, lack a guard bit for their REAL*10 format though they do conform to IEEE standard 754 in REAL*8 and REAL*4 .

Seymour Cray's floating-point arithmetics on the CDC 6600 and Univac 1107 and their descendants required extra operations to be

compiled after every add/subtract to achieve the effect of a guard bit; since that effect hardly ever mattered much, hardly any compilers complied, so most users could truthfully say that their arithmetics lacked a guard bit. That is one reason why neither of his earlier machines is listed in Table 2 .

A Spurious Backward Error-Analysis

Since so many computers, lacking a guard bit, occasionally get a difference D := A - B that is utterly different from A - B , their manufacturers must have a rationale of sorts to excuse that discrepancy. Here it is.

Review the explanation above of the error in D . Note that the bit dropped from B can be regarded as subtracted or added into a position just past the last bit of A . Hence D := A - B yields actually $D = A^{\sharp} - B$ wherein $A^{\sharp}$ differs from A only in bits beyond the last bit stored for A . Since the last few bits of A are almost never accurate, changing the very last is no big deal.

A little algebra turns all this into mathematics. Computers asked to compute D := A - B generally compute $D = (A - B)(1 + d)$ instead, where the unknown perturbation d due to roundoff must satisfy $|d| \le r$. Here r is the roundoff threshold mentioned in Table 3 and the Risk Equation above. Machines that lack a guard digit compute something else; their $D = A(1+a) - B(1+b)$ in which the perturbations a and b due to roundoff must satisfy $a\,b = 0$ and $|a| + |b| \le r$. Since operands A and B inherit uncertainty from previous operations, perturbing them a bit can't hurt the computation of a continuous function. Besides, the two ways to compute D differ significantly only when A and B are so close that massive cancellation occurs in A - B and, as we all know, cancellation portends numerical instability.

Like every believable lie, the foregoing argument is pernicious because it is mostly true. Cancellation is often identified as a culprit in numerically unstable algorithms, but not always; and often cancellation is the goal of a computation, as when linear equations are solved, so cancellation cannot always be bad. Most numerically stable algorithms are indeed insensitive to end-figure perturbations in their intermediate results, but not all of them. AREA is a counter-example. AREA is accurate, except on CRAYs, despite massive cancellation in A-B or C-D , and it fails on a CRAY just because of end-figure perturbations to A, B, C or D .

Evidently the familiar arguments about cancellation and end-figure perturbations are flawed. Exceptions like AREA and others to be discussed later are rare, rare enough to come as surprises, but not rare enough to ignore. If we hope to discern the difference between stable and unstable algorithms, and to diagnose correctly the latters' diverse failure modes, we need better ideas.

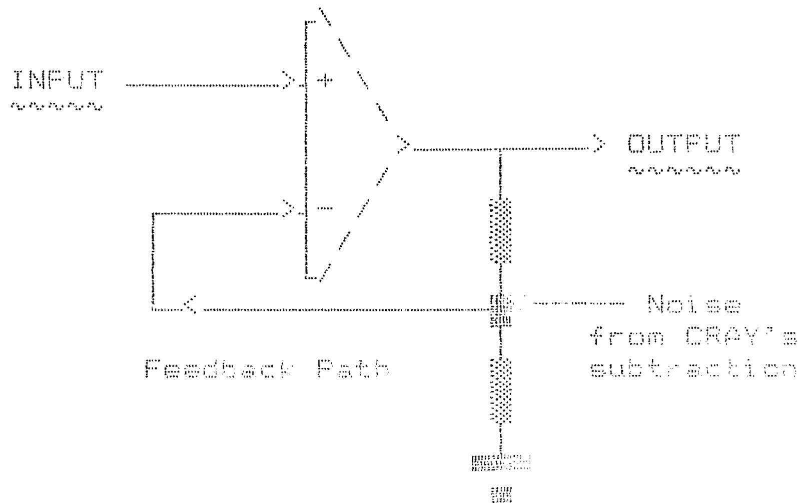


Figure 1 : Noise leaks into the Feedback Path

One way to visualize the distructive effect of a missing guard bit upon some otherwise stable algorithms is to consider the effect of a noisy connection in the feed-back loop of a control system shown in Figure 1. The noise obscures a tiny difference between some function of the output and its intended goal just as the missing guard bit obscures a difference that would have cancelled cleanly otherwise; then the noise is fed back and amplified enormously along with the input signal. But this is all too figurative; we have to return to mathematics to see what is really happening.

Benign Singularities

What causes a numerical program to fail is approaching too near a singularity or passing beyond it. Arithmetic idiosyncracies like lack of a guard digit do not alter this fact, but they do change the meaning of "singularity." Three examples will show how.

Some singularities are *removable*; an instance is at $x = 0$ in

$$S(x) := \text{if } x = 0 \text{ then } 1 \text{ else } \sin(x)/x \,.$$

Mathematically, this conditional statement defines a function S that is analytic at every finite x ; the singularity at $x = 0$ has been removed by the test and branch. Computationally, the same is true provided the library's implementation of sin is as relatively accurate as on almost all machines listed in Table 2. When both numerator and denominator of $\sin(x)/x$ are accurate to nearly full precision, the same must be true of their quotient.

And the same would be true on a CRAY, using an implementation of sin described by CRAY's Jim Kiernan at this CUG meeting, but for a tiny technicality; what a CRAY computes for $\sin(x)/x$ is actually $(1/x)\sin(x)$. If x is nonzero but too tiny -- nearly CRAY's underflow threshold -- its reciprocal will overflow before CRAY's computatation of $(1/x)\sin(x)$ has been completed. To repair that defect, replace the conditional statement above by

$$S(x) := \text{if } 1 + |x| = 1 \text{ then } 1 \text{ else } \sin(x)/x \,;$$

this removes the computational singularity on all machines I know.

(S(x) could still fail on machines that round by "jamming", as John von Neumann recommended, because they round $1 + |x|$ away from 1 for all nonzero x . IIT 2C87 and 3C87 chips round their REAL*10 format this way, but S(x) works anyway because of a quirk in their comparison operator to which we shall return.)

A singularity at $y = 1$ disappears similarly from the definition

$$L(y) := \text{if } y = 1 \text{ then } 1 \text{ else } Ln(y)/(y - 1) .$$

L(y) is analytic at all $y > 0$ including near $y = 1$ where its Taylor series is

$$L(y) = 1 - (y-1)/2 + (y-1)^2/3 - (y-1)^3/4 + \dots .$$

We need not use the series in our program; the definition above is fully accurate despite cancellation near $y = 1$ provided that, when y is regarded as known exactly, nonzero computed values of Ln(y) and $y - 1$ are accurate to nearly full precision, as they are on all machines in Table 2. The thought that inherited error in $y = 1 + junk$ could leave $y - 1 = junk$ and produce a result $L(y) \doteq junk/junk$ should not distract you; *junk* nearly cancels, leaving $L(y) \doteq 1 - junk/2$ to inherit no more error from *junk* than it deserves. Our simple definition is numerically stable.

Except on CRAYs. Jim Kiernan's logarithm is accurate enough but CRAY's subtract is not; review the 8-bit example A - B but with its binary point moved three places left. To prevent CRAY's subtraction from reintroducing singularity, rewrite L(y) thus:

$$L(y) := \text{if } (y-0.5) = 0.5 \text{ then } 1 \text{ else } Ln(y)/((y-0.5) - 0.5) .$$

This pornographic expression is accurate to nearly full precision on all computers I know provided the run-time library's Ln(...) is accurate, and provided the denominator is not compiled back to " (y - 1) " by the kind of over-zealous "optimization" practised by an old Univac Fortran compiler.

The third example is taken from real life with all its complexity. Financial computations often entail the removal of a singularity in the computation of the present or future value of a sequence of N constant discounted or interest-garnering cash flows;

$$V(N, i) := \text{if } i = 0 \text{ then } N \text{ else } ((1+i)^N - 1)/i .$$

Here i is the discount or interest rate expressed as a fraction; expressed as a percentage the rate would be $I = 100\,i\,\%$. On no computer is this conditional statement numerically stable; all accuracy can be lost if $|i|$ is tiny enough. Redefining V as follows removes much of the computational singularity:

V(N, i) :
    y := 1 + i ;
    Return V := if y = 1 then N else $(1 - y^N)/(1 - y)$.

Provided N is an integer, positive or negative, and provided y^N is computed in a reasonably accurate way either by repeated squaring (commonly used for Fortran's Y**N) or by a formula like exp(N Ln(y) (universally used in Fortran's Y**REAL(N))) , the foregoing redefinition of V is provably accurate to at least half the sig. digits carried by the arithmetic of any computer in Table 2. (Of course, almost all sig. digits carried are correct unless interest rate i is too tiny to be seen often nowadays.)

Here is a glimpse at the proof of V's claims to accuracy. Start from an observation that $y := 1 + i$ actually yields $y = 1 + i'$ where $|i - i'| < u = 1$ ulp of y ; here an *ulp* is one *u*nit in the

*l*ast *p*lace stored. When i is so small as jeopardizes accuracy, i^2/u must be an integer, which presages an interesting proof.

That proof is invalid on CRAYs. However another more interesting proof guarantees similar accuracy claims for CRAYs' V provided y^N comes either from Jim Kiernan's new Y**X with X = REAL(N) or from Y**N via repeated squaring but not using "chopped" **F* multiplication on a CRAY X-MP or Y-MP, about which more later. Why can't CRAYs' computed denominator $1 - y$ be as utterly wrong for V as it can be for the first version of L(y) ? Because if $y := 1 + i$ is slightly less than 1 the last sig. bit of y turns out to be 0 so the denominator is computed exactly.

Whether our tricky program for V(N, i) deserves to be considered numerically stable makes a good subject for debate. On a CRAY 2 V is accurate to at least seven sig. dec., probably good enough for government work. Among financial calculators and software packages on personal computers, only the best do better. (How? See pp. 62-5 of the *Apple Numerics Manual, Second Edition* (1988) Addison-Wesley, Calif.) And yet our programs for V and L will make many a reader uneasy, as if he were tottering on the brink of numerical instability. So many rules are being violated! The programs fly in the face of a familiar maxim,

NEVER test floating-point numbers for equality;

they flirt with subtractive cancellation; they compute *junk/junk* and they get away with it. Don't they deserve to fail on CRAYs ?

Confused Comparisons on the CRAY2

Let us try to be logical about comparing floating-point numbers. We can agree that numerical software is often better served by a boolean function like eq(x, y, *tol*) , which is *True* just when $|x - y| \le tol$, than by the simpler looking predicate " $x = y$ ", provided a suitable choice for *tol* can be found. Nevertheless, whether to cope with the case $tol := 0$ or to implement eq(...) itself, a computer system still needs a consistent family of familiar predicates $=, \ne, \le$, etc. Responsibility for consistency is shared by implementers of the compilers and of the hardware. Which of these, if any, deserve blame when a statement like

Df := if $x = y$ then f1(y) else $(f(x) - f(y))/(x - y)$

emits at run-time a paradoxical error-message like

" *ERROR*: Division by 0.0 , ..."

or should the programmer be blamed for not using eq(x, y, *tol*) ? Choosing *tol* is the programmer's problem. Our problem is this:

How can $x \ne y$ and $x - y = 0.0$ simultaneously?

It can happen in three ways:

1. Computed $x - y$ underflows to 0.0 .
2. Computed $x - y$ really is nonzero but the divider dissents.
3. Computed $x - y = 0.0$ though x and y are big and different.

None of these things can happen to computer systems that conform faithfully to IEEE standards 754/854. All can happen to CRAYs.

The first two ways occur only very rarely, though often enough to undermine the serenity of conscientious programmers. The first afflicts all computers that flush $x - y$ to 0.0 when x and y are bigger than the underflow threshold but their difference is

not. The second occurs when a CRAY computes not .../(x-y) but (...)*(1/(x-y)) and the reciprocal overflows prematurely. (The second way also afflicted descendants of the CDC 6600 because its divider and multiplier examined only 12 leading bits to test for 0.0 instead of the necessary 13 tested by the adder.) We might wish that programs could detect over/underflow and take some corrective action to cope with these situations, but no CRAY has hardware to signal underflow to software.

The third way to get 0 can happen only on a CRAY 2 , on a CDC 6600's descendant, or on a cheap handheld calculator, caused by lack of a guard digit. Recall computing zero for D := A - B above using 8-bit arithmetic like a CRAY 2's. This phenomenon cannot be ignored; it undermines divided differences, stopping criteria for iterations, terminations of trajectories, avoidance of singularity, and estimates of arithmetic granularity. Let us return to them after closing what looks like an escape hatch.

The paradoxical error-message above could be precluded altogether if compilers treated every appearance of the predicate " x = y " as if it had been written " x-y = 0.0 ". Compilers for the CDC 6600 used to do that; it was a cure worse than the disease. It ruined the first property that might reasonably be expected from the equality predicate, namely that ...

  y = x should imply f(y) = f(x) for every continuous f .

(*Discontinuous* functions like f(x) := 1/x are allowed to violate f(0) = f(-0) despite that 0 = -0 on a machine that has both.) This reasonable expectation obliges all language implementors to translate the predicate y = x into the machine instructions that compare y with x bitwise rather than subtract them on a CRAY 2 ; the same happens on other CRAYs (though motivated perhaps by the deplorable practice of packing character strings into REAL variables). Therefore no reasonable expedient in the compiler can preclude that paradoxical error message alleging division by zero; instead programmers must insert their own defensive tests.

Things could be worse. Comparison operations built into the IIT 2C87 and 3C87 allege equality of REAL*10 (but not REAL*8 nor REAL*4) operands whose computed difference can be as big as 3 ulps. Would-be portable numerical software intended for all the machines mentioned so far, allowing also for compilers' vagaries, has to include tests and branches as redundantly paranoid as those shown nearby in the allegedly transportable program NEXT1(X) .

NEXT1(X) finds the closest floating-point neighbor to 1.0 on the same side as X . RATAREA used it to assess the granularity of floating-point numbers while creating critical data to probe how accurately AREA ran on whatever machine was under test. It has other uses too, as we shall see when NEXT1 appears again.

Few programmers write programs like NEXT1 , and fewer are so paranoid as to expect a computer to lie about whether y = x ; so among the numerous programs analogous to NEXT1 buried deeply in widely distributed packages of portable numerical software may well be some that behave strangely on a CRAY 2 or IIT chip.

NEXT1 Programmed Transportably in a Nondescript Language:

```
Real Function  NEXT1( Real Variable  X ) :
    ...  := the number next to  1.0  on the same side as  X .
    ...  NEXT1(2.0) and NEXT1(0.5)  are believed to work right on
    ...  all floating-point hardware,  aberrant or not,  including
    ...  CRAYs,  IIT 2C87 and 3C87 10-byte format,  CYBERs, .... .
    ...  The argument  X  will be left unchanged.

    ...  First we check  Decimal-Binary Conversion  of  0.5 :
    H := 0.5 ;  U := H+H ; ...  We hope  H = 1/2  and  U = 1 .
    If  ( U*U-H ≠ H ) or ( (U*U-H)-H ≠ 0.0 )  then  {
        Protest("Decimal-Binary Conversion of  0.5  is Suspect.");
        Stop  } ;

    ...  Distrustfully,  we obtain two estimates of  NEXT1 :
    ...  Z  is the first,  obtained by repeated averaging with  1.
    Y := X ;  Z := U ;
    While  ( (Z-H ≠ Y-H) or ((Z-H)-(Y-H) ≠ 0.0) )  and
           ( (Y-H ≠ H) or ((Y-H)-H ≠ 0.0) )
        do {  Z := Y ;  Y := H*Z + H  } ;

    ...  A 2nd estimate  S  comes from a dwindling difference  D.
    Y := X ;  S := U ;  D := (Y-H) - H ;
    While  ( (S-H ≠ Y-H) or ((S-H)-(Y-H) ≠ 0.0) )  and  (D ≠ 0.0)
        do {  S := Y ;  Y := (D*H + H) + H ;  D := (Y-H) - H  } ;

    ...  Check that  S = Z ≠ 1  unless  X = 1 :
    If  ( (Z ≠ S) or (Z-S ≠ 0.0) )  or
        ( ((X-H) - H ≠ 0.0) and ((Z-H) - H = 0.0) )  then  {
        Protest( "Something is wrong with values computed for" );
        Protest( " NEXT1(", X, "),  namely"                    );
        Protest( " Z =", Z, "  and   S =", S, " ."              );
        Protest( "Check how faithfully your compiler's output" );
        Protest( "matches this source-code.  Parentheses must" );
        Protest( "be respected,  and assignments not bypassed" );
        Protest( "by retention of variables in extra-precise"  );
        Protest( "registers.  If your compiler is unfaithful," );
        Protest( "try replacing arithmetic expressions by"     );
        Protest( "equivalent but separately compiled function" );
        Protest( "subroutines to thwart would-be optimization" );
        Protest( "by the compiler.  If this message reappears,");
        Protest( "your computer's floating-point arithmetic"   );
        Protest( "is so strange that I want to know about it." );
        Protest( "Please contact me:    Prof. W. Kahan"        );
        Protest( "                      E.E. & Computer Science");
        Protest( "                      Univ. of California"    );
        Protest( "(510) 642-5638        Berkeley  CA 94720 "    );
        Stop  } ;
    Return  NEXT1 := Z ;  End NEXT1 .
```

Divided Differences
The singularities removed from S(x), L(y) and V(N, i) above are typical for *Divided Differences*; for instance,

$$D\sin(x, 0) = S(x) \quad \text{and} \quad D^2\cos(-2x, 0, 2x) = -S(x)^2/2$$

where we define the *First Order Divided Difference*

```
    Df(x, y)  :=  if  y = x  then  f'(x)   ... First derivative
                                else  (f(x) - f(y))/(x - y)
```

and the *Second Order Divided Difference*

```
   D²f(x, y, z)  :=  if  z = y = x  then  f"(x)/2   ... Second derivative
                     else if  z = x  then  (f'(x) - Df(x,y))/(x-y)
                     else  (Df(x,y) - Df(y,z))/(x-z) .
```

They wallow in the naughtiness we have come to distrust on CRAYs: floating-point comparison, subtractive cancellation, *junk/junk* .

It is a pity that they are ubiquitous in numerical computation. They figure in the solution of differential equations, prediction and extrapolation, optimization, geometrical computation, and iterative solutions of nonlinear equations, among other things. These are a CRAY's bread and butter so it has to compute divided differences safely.

Safety is not a big issue yet on CRAYs. If occasionally some software imported from other machines has failed on CRAYs, it has been rewritten by clever programmers whose salaries add little to unavoidable overheads associated with world-beating computers. If the rewritten code has been obscure or pornographic, nobody else has cared who did not have to read it. Or so we have hoped.

How much more has to be paid to develop obscurantist software for CRAYs than to import eqipollent software from a larger community of computers like those in Table 2 ? Divided differences provide a case study for this question. Let us consider how much extra thought a programmer has to put into them after he discovers what a CRAY will do to them.

Df(x, y) blooms in an environment that tends to drive x and y together, and when they get close the denominator x - y is computed exactly on other machines; on a CRAY it is as if x or y had been altered in bits beyond the last one stored. Meanwhile f(x) and f(y) approach each other too; generally their computed values are obscured by roundoff, so Df(x, y) must approach *junk*/(x-y) on other machines, *junk/junk* on a CRAY. Not much to choose between those. Of course, this cannot be the whole story because it does not explain why L(y) and V(N, i) are so accurate as they are; but it is a plausible story so far.

One application of Df(x, y) is to solve a nonlinear equation f(x) = 0 by *Secant Iteration*: $x_{n+1} = x_n - f(x_n)/Df(x_n, x_{n-1})$. This iteration is preferred when no explicit formula exists for a derivative needed in *Newton's Iteration* $x_{n+1} = f(x_n)/f'(x_n)$. Otherwise the iterations are similar, comparable in speed and global convergence properties, except for their vulnerability to roundoff.

To prevent the secant iteration from dithering or worse, it should be stopped as soon as the computed value of $f(x_n)$ has become smaller than its uncertainty due to roundoff; how to estimate its uncertainty will be discussed later. Stopped that

way, the iteration can be proved to converge to a simple root of the equation about as closely as the equation determines the root when uncertainty in f is taken into account. Except on a CRAY.

On a CRAY 2 roundoff can diminish $(x_n - x_{n-1})$, causing the next iterate x_{n+1} to fall short; this can slow the iteration or stop it prematurely. On an X-MP or Y-MP roundoff tends to expand $(x_n - x_{n-1})$, causing the next iterate x_{n+1} to overshoot; this can make the iteration dither or, if convergence was expected to be monotonic, stop it prematurely. These extremely unlikely effects can be significant only when the graph of f is steep and its uncertainty very small, as when the zero of f comes very close to a pole, in which case the equation determines its root more sharply than the CRAY can be expected to find it without extra logic in the zero-finder to force an extra iteration or two.

Striving for best possible results, the conscientious programmer will have to fuss over that extra logic. His fuss can improve the computed root only if it differs from a power of 2 in just its last few bits, and those are the only bits he can improve.

Most users of CRAYs strive for results that are just good enough rather than best possible. One of them who has to compute, say, a divided difference $D^2f(x,y,z)$ will test differences among its arguments x, y, z against tolerances that will probably separate them by at least thousands of units in their last places. Then the computed value of $D^2f(x,y,z)$ will be obscured by errors of two kinds. The first and most important will be inherited from error in the data f(...) , and can be estimated from a practical understanding of the data and its application. The second kind will arise from the process of differencing and dividing, and will depend upon the computer's arithmetic; this kind of error is the kind a programmer prefers to ignore. Dare he ignore it?

Provided y lies strictly between x and z , *but no matter how close*, the differencing and dividing will contribute no more new uncertainty to D^2f than if the data values f(...) had each been perturbed by a unit or two in the last place of its stored value. Consequently, the uncertainty in D^2f is preponderantly what inheritance from f(...) deserves; subsequent roundoff can be ignored, and a little algebra proves it. Except on CRAYs.

Because CRAYs lack a guard bit, their differencing and dividing adds more error than described in the last paragraph whenever two adjacent arguments x, y, z closely straddle a power of 2 . The effect is roughly as if the values of f were perturbed by $\pm df/N$ where df is the change in f across the straddle and N counts how many floating-point numbers lie inside the straddle. Proving this claim for CRAYs is rather more complicated than proving the previous paragraph's neat error bounds for other machines, and interpreting $\pm df/N$ is far more complicated too.

Do Physicists ever wonder whether faint creases that flicker in certain wave-fronts or in fields with steep but steady gradients computed on CRAYs are due to something other than Physics?

The Pessimism of *Almost² Right* Error Analysis on a CRAY
Error analysis tends to be pessimistic, and more so on CRAYs than other machines, and you pay for that pessimism when you pay to develop reliable numerical software. Pessimism comes less from the error-analyst's dour personality than from his mental model of computer arithmetic. At its most extreme, pessimism can condemn a program that actually works satisfactorily on a CRAY, causing it to be replaced by something worse. If you believed subtractive cancellation must produce useless *junk/junk* , you would have had to discard L(y) and V(N, i) above. A more striking example is A(Z) presented below; it will contain no subtractions at all.

First we examine the mental models of roundoff error analysts use, to see why they are more pessimistic for CRAYs than for other machines. The models interpret the statement " D := A⊛B " as if it actually computed either

$$D = (A \circledast B)(1+d) \quad \text{or} \quad D = A(1+a) - B(1+b) ,$$

wherein ⊛ is one of the binary rational arithmetic operators

$$+ , \; - , \; * \; \text{ or } \; / ,$$

and a , b and d represent the effects of roundoff, unknown but no bigger than a known roundoff threshold r . (See Table 3.)

The first formula, the one with d in it, is a model devised by W. Givens, J. H. Wilkinson, F. L. Bauer and numerous others in the first decade of computing as we know it, 1947 - 57. It works for all operations on all machines listed in Table 2 , and for multiplication, division, and addition of magnitudes on CRAYs.

The second model, the one with a and b in it, was devised in the mid 1960s to cope with subtraction of magnitudes on machines like Univacs, CDC's, and now CRAYs, that lack a guard digit.

The first model has been spectacularly successful at explaining the behavior of numerical software and then motivating development of superior algorithms we now take for granted. So many of its successes, especially with matrix computations, have carried over to the second model that many people believe the models to be practically indistinguishable. That belief may have something to do with the widespread acquiescence to a similar model promulgated by W. Stan Brown in *ACM Trans. on Math. Software* **7** (1981) pp. 445-80, which has affected ADA profoundly and influenced recent revised standards for C and Fortran. That belief is wrong.

The first model is far more powerful and less pessimistic than the second. For instance, arithmetic conforming to the first model is good enough to compute a second divided difference D^2f as if its data f(...) had been perturbed only in their last digits; a CRAY cannot do that, as we just saw above, although it conforms to the second model. Assuming that Ln(...) is accurate to full precision, the first model implies full accuracy for the simpler version of L(y) above, but the second model implies accuracy for neither the simpler nor the pornographic versions of L(y) .

Perhaps the second model's pessimism comes from having two error terms a and b instead of one d . Whatever the reason, error analyses based upon the second model tend to over-estimate actual errors by bigger factors than do analyses based on the first.

For all its flaws the second model provides programmers with a far simpler description of CRAY's subtraction than do its manuals. The vast majority of applications need nothing more. And a pithy euphemism summarizes the second model:

Almost² Right: If you wish to compute $f(x, y, z, \dots)$ on CRAYs you should instead be satisfied to get $F(X, Y, Z, \dots)$ where each of $X, Y, Z, \dots$ differs respectively from $x, y, z, \dots$ by at most a unit in its last place, and $F(X, Y, Z, \dots)$ differs likewise from $f(X, Y, Z, \dots)$. In other words, you should be satisfied with an almost right result for data that are almost right. That is what "Almost Almost Right" means.

For a function f continuous in all its arguments, this looks like a good deal, so good that many a computer professional still believes that *Almost² Right is Good Enough*, though experienced error analysts know why something is wrong with Almost² Right.

Jim Kiernan knows it too; that is why he rewrote CRAY's library of elementary functions. Many of them used to be at best Almost² Right, and consequently spoiled the accuracy of otherwise correct software. For instance, if Ln(...) were merely Almost² Right, both versions of $L(y) := Ln(y)/(y-1)$ above would be inaccurate for y close to 1 . The reason is that $Ln(y)$ would change to at best $Ln(Y)$ for some Y almost equal to y , and that would change $L(y)$ by roughly $(Y-y)/((y-1)y) + O(Y-y)^2/(y-1)$, which is almost as big as $L(y)$ when y and Y both almost equal 1 .

Arithmetic and elementary functions no better than Almost² Right impede efficient computational removal of mathematically removable singularities. For example consider the function $A(z)$ defined in a short program shown nearby above its graph in Figure 2 . A removed singularity at $z = 1$ leaves it analytic for all $z > 0$, and the program is accurate on all machines in Table 2 and most likely all others too provided their libraries implement arccosh, arccos and Ln accurately. (An accurately implemented arccosh based upon our earlier program $L(y)$ accompanies $A(z)$.)

If arccosh, arccos and Ln are only Almost² Right, or if you think they are, you cannot trust Version 1 of $A(z)$. Instead you must turn to Version 2. It uses a *Taylor Series*

$$A(1+t) = 1/2 - t/6 - t^2/20 + 124t^3/945 - 8221t^4/113400 - \dots$$

to remove the singularity from an interval $1-T < z < 1+T$. Here the threshold T is designed to minimize the worst of two errors: one is from Almost² Right functions, and resembles the error due to $Y-y$ in $L(y)$ above; the other comes from using just three terms of the Taylor series. The reader is left to wonder where the series came from -- its derivation cannot be trivial if 8221 is a prime --, and whether $r :=$ NEXT1(2.0) - 1 is a fair bound for the perturbations $Z-z$, $Y-y$, ... in Almost² Right functions, and why T takes two square roots, and how much time he would have to spend to find out.

Unfortunately, Version 2 must lose a quarter of the significant bits carried by the arithmetic; it loses the last 12 bits on a CRAY . Yet with Jim Kiernan's new library codes, and with the arccosh program here, Version 1 of $A(z)$ was fully accurate.

Whoever insists that arithmetic and elementary functions are good enough if Almost² Right must occasionally put up with numerical software gratuitously more complex, more costly, and inaccurate.

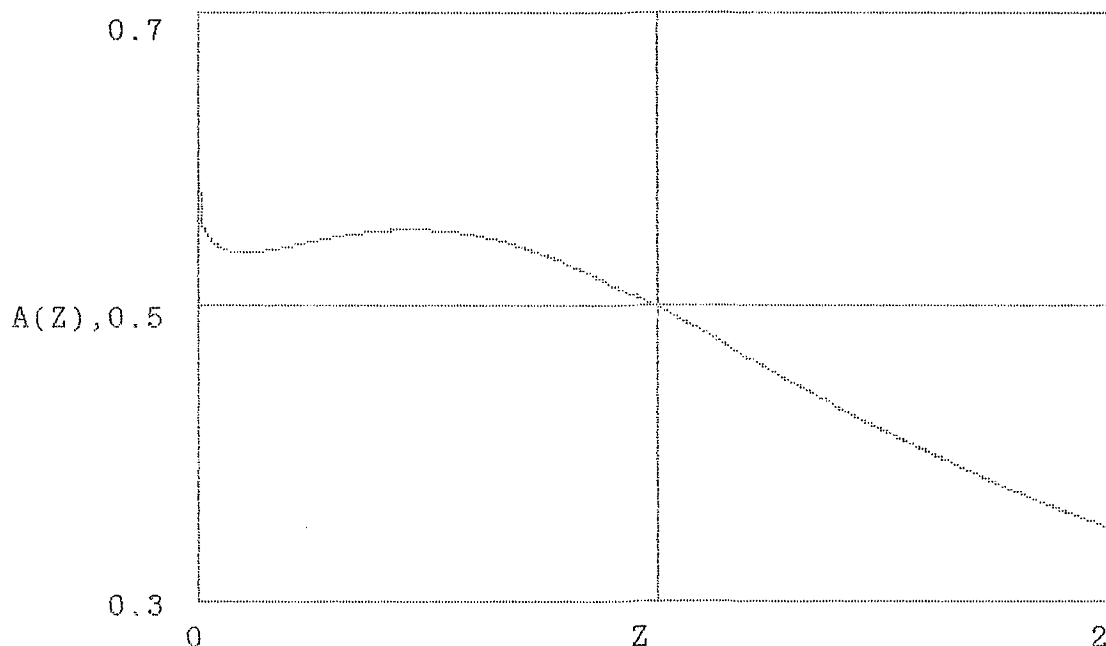
Programs to Compute A(z) and arccosh
Written Ostensibly Transportably in a Nondescript Language

```
Real Function  A( Real Variable  Z ) :=
    if  Z < 1  then  -arctan(Ln(Z))/arccos(Z)²
              else if  Z = 1  then  0.5
                              else  arctan(Ln(Z))/arccosh(Z)² ;
    End A . ...  Version 1,  probably accurate on all machines.


Real Function  A( Real Variable  Z ) :
... Version 2  is less trusting and less accurate than  V. 1.
    Constant  T := SQRT(SQRT(8*(NEXT1(2.0) - 1))) ;
    Return  A := if  Z < 1-T  then  -arctan(Ln(Z))/arccos(Z)²
            else if  Z < 1+T  then  0.5 - (Z-1)/6 - (Z-1)²/20
                              else  arctan(Ln(Z))/arccosh(Z)² ;
    End A . ...  Version 2.


Real Function  arccosh( Real Variable  Z ) :
... Good enough for CRAYs with a good version of L(..), q.v.
    U := SQRT((Z-0.5)-0.5)*(SQRT((Z-0.5)-0.5) + SQRT(Z+1)) ;
    Return  arccosh := U*L(U+1) ;  End arccosh.
```

Figure 2 :



Compensated Summation
As N approaches infinity, the accuracy with which a sum

$$S_N = X_0 + X_1 + X_2 + \ldots + X_{N-1} + X_N$$

can be computed is threatened by a singularity. One way to think of it is as a singularity at $1/N = 0$. An ancient computational technique exists that removes this singularity without introducing any threshold that would prevent $1/N$ from coming arbitrarily close to 0 for practical purposes; but it can fail on a CRAY.

This all seems so far-fetched that at least one application should be introduced before the technique is described. Consider *initial value problems*; given $\mathbf{y}_0$ and $\mathbf{f}(\ldots)$ find $\mathbf{y}(t)$ satisfying

$$\mathbf{y}(0) = \mathbf{y}_0 \quad \text{and} \quad d\mathbf{y}(t)/dt = \mathbf{f}(\mathbf{y}(t)) \quad \text{for all} \quad t > 0 \ .$$

The solution to this problem could predict the orbit of a planet or satellite, the trajectory of a cannon ball, the state of an oscillating mechanism or electrical circuit, the propagation of a travelling wave down a channel or electrical stripline, ...

A typical numerical method *discretizes* this problem to compute an approximation $\mathbf{Y}(t)$ to $\mathbf{y}(t)$ as follows:

First a formula $\mathbf{F}(.., ..)$ is selected, then a stepsize $h > 0$ with which to cover a specified interval $0 \le t \le H$. To simplify notation we shall pretend that h is constant and that $N = H/h$ is the number of steps needed to cover the t-interval. (Actually h could vary so long as it never got excessively bigger than the average stepsize H/N .) Therefore we treat N as the parameter at our disposal instead of h .

The construction of $\mathbf{Y}(\ldots)$ begins with initialization,

$$\mathbf{Y}(0) := \mathbf{y}_0 \ ; \quad t := 0 \ ;$$

and then repeats iteratively the computation

$$\mathbf{Y}(t+h) := \mathbf{Y}(t) + h*\mathbf{F}(\mathbf{Y}(t),\ h) \ ; \quad t := t+h \ ;$$

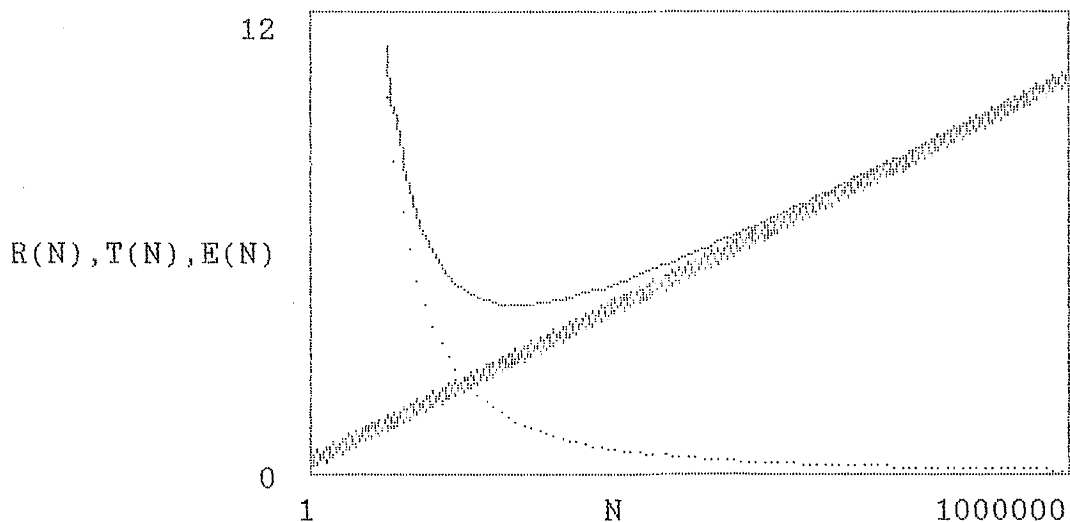
a total of N times, ending with $t = H$ and $\mathbf{Y}(H)$ in hand. We call this a *Trajectory Calculation* regardless of whether the whole trajectory, a sequence of pairs $\{t,\ \mathbf{Y}(t)\}$, or just its end $\{H,\ \mathbf{Y}(H)\}$ is the desired result. Think of the computed trajectory as a sequence of dots close to the true trajectory, a curve traced out by $\{t,\ \mathbf{y}(t)\}$. How close?

The error $\mathbf{Y}(H)-\mathbf{y}(H)$ has three constituents of which two depend upon N and $\mathbf{F}(.., ..)$. The first constituent is inherited from the error in $\mathbf{y}_0$ and $\mathbf{f}(\ldots)$; this is physically relevant error compared with which we hope to make the rest negligible. Second is the *Truncation* error $\mathbf{T}(N)$, which is what the error would be if $\mathbf{y}_0$ and $\mathbf{f}(\ldots)$ were exact and no rounding errors occurred. Usually $\mathbf{T}(N) = O(1/N)^P$, as $1/N \to 0$, for some *order* $P \ge 1$ that depends upon how well the formula $\mathbf{F}(\mathbf{Y},\ h)$ samples $\mathbf{f}(\ldots)$; a higher order goes with a (perhaps *Runge-Kutta*) formula that approaches a suitable average of $\mathbf{f}(\ldots)$ faster as $h \to 0$.

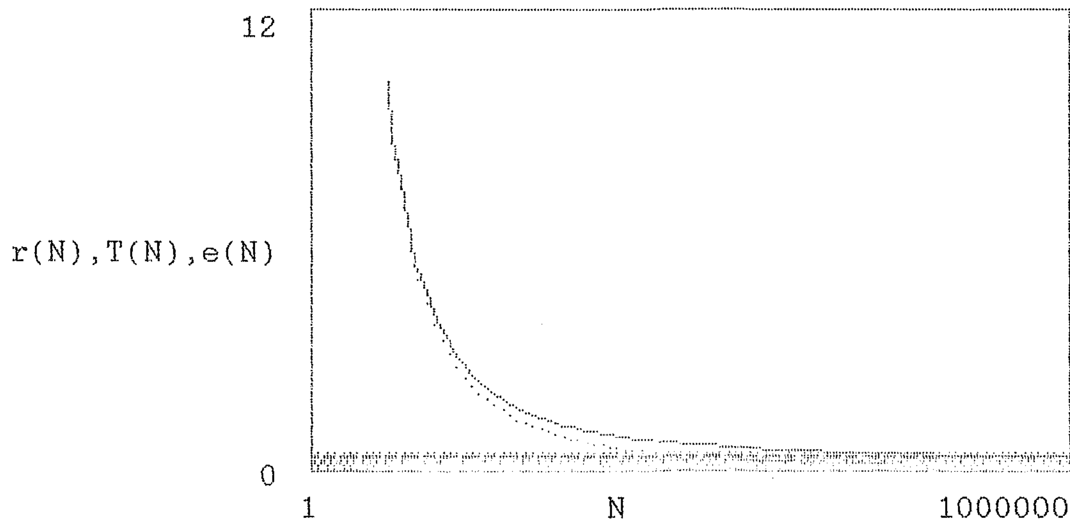
The third error constituent $\mathbf{R}(N)$ is due to roundoff. For the kind of trajectory we have in mind, neither exponentially stable nor exponentially unstable, $\mathbf{R}(N)$ approaches infinity with N roughly as shown in the first graph in Figure 3 . Graphs like that appear in some texts to explain how to minimize the total error $\mathbf{E}(N)$ by choosing h or $N = H/h$ " optimally." Bunk!

Figure 3 : **ERROR IN TRAJECTORY CALCULATIONS**

With Conventional Uncompensated Summation



With Compensated Summation



Legend:
~~~~~~

	N	=	No. of time-steps to trace Trajectory
		=	(Time to trace Trajectory)/(time-step)
......	T(N)	=	Truncation error due to discrete steps
xxxxxx	R(N)	=	Rounding error in Conventional Summation
	r(N)	=	Rounding error in Compensated Summation
_____	E(N)	=	T(N) + R(N) Total trajectory error
	e(N)	=	T(N) + r(N) for each summation method
~~~~~~

Most of the roundoff in R(N) comes from the additions marked **+** in the iterative computation, and most of that can be suppressed by *Compensated Summation*. This was part of the *Runge-Kutta-Gill* method devised by S. Gill in the late 1940s for his Cambridge thesis. His method worked in fixed-point arithmetic; versions suitable for floating-point were published simultaneously in 1965 by R. Møller in *BIT* and by me in *Comm. ACM.* The idea was to capture the rounding error in each iteration for feedback into the next as a correction **C** thus:

Initialization: $\mathbf{Y}(0) := \mathbf{y}_o$; $\mathbf{C} := \mathbf{0}$; t := 0 ; c := 0 .

Iterative loop repeated N times:

```
        dt := h + c ;                DY := h*F(Y(t), h) + C ;
        newt := t + dt ;             Y(t+h) := Y(t) + DY ;
        c := (t - newt) + dt ;       C := (Y(t) - Y(t+h)) + DY ;
        t := newt .                  DON'T OMIT PARENTHESES !
```

The cost of this compensated summation is trifling: three extra add/subtracts per component computed in the loop, and an extra array to store **C** . Its effect is shown in the second graph in Figure 3. The contribution **r**(N) of roundoff to total error is reduced to something practically independent of N provided 1/N significantly exceeds the roundoff threshold *r* of Table 3.

How much is compensated summation worth? That depends; it is valuable only when N is huge, the accuracy desired is near the limit achievable by the computer's arithmetic, and the trajectory is neither exponentially stable (it would forget all errors but the last few) nor exponentially unstable (only the first few errors would matter). Compensated summation is invaluable for celestial mechanics, worthless for switching circuit simulation.

Among other applications for compensated summation are adaptive quadrature requiring unpredictably huge numbers of samples of the integrand to cope with nearly improper integrals, and slowly convergent series requiring an unpredictably large number of terms to converge accurately enough. Another application arises in languages that allow a REAL variable to control a **for** loop; that variable, like the trajectory's t , has to be incremented accurately lest the loop stop prematurely or overshoot its limit.

A proof that compensated summation works needs no more than the first model of roundoff mentioned five pages back; see D. E. Knuth's *The Art of Computer Programming* vol. 2, *Seminumerical Algorithms* 2nd ed. (1981) Addison-Wesley, Calif., problem 19, p. 229 and p. 572-3. Therefore compensated summation works with all the machines in Table 2 and even with strange arithmetics that represent numbers internally by their logarithms. But it fails on CRAYs. Extremely rarely.

A program CMPSUM shown nearby contrives a series S_N for which compensated summation fails on a CRAY but, of course, on no machine in Table 2. The number of additions is N = 83L , to be chosen at run-time. Results for L = 1000000 appear in Table 5.

CMPSUM: a Program to Sum a Long Series
Programmed Ostensibly Portably in a Nondescript Language

```
Program  CMPSUM:
... compares three ways to sum a series of   83*L + 1   terms
...            S  =  X0 + X1 + X2 + ... + X83L
... in which  L  is a large integer entered from the keyboard.
... SS  is the  Simple Sum  computed in the obvious way.
... SC  is a  Compensated Sum  computed more accurately. ( ? )
... To compare the accuracies of the two methods of summation,
... they are compared with the true sum  S  computed exactly.
... The terms  Xj  are contrived to produce striking results.
... Errors are measured in  ulps ;  an ulp is a unit in the
... last sig. digit carried by the computer's floating-point.

Real  A..H, O..Z ; ...  Choose your floating-point precision.
Constants  H = 0.5 ,  One = 1.0 ,  Two = 2.0 ,  At = 80.0 ;
Long Integers  I..N ; ...  Fortran  convention for integers.
Constant  Lmax = 16777215 ; ...  the limit for a  CRAY X-MP.

Display( "Key in a positive integer  L  that will" );
Display( "  determine how long the program runs."  );
Keyboard Input  L ;  if  L ≤ 0  then  Stop ;
If  L > Lmax  then  Stop ; ...  too big for a  CRAY X-MP.

... Find  E = one ulp,  the computer's level of roundoff:
E := (H - NEXT1(H)) + H ; ...  = 1.0 - 0.999999... .
Display( "This computer's  One ulp  E = ", E );

... Initializations:
SS := One + At*E*L ;  SC := SS ;  S1 := SS - One ;
Xz := Two*E ;  Xy := S1 + Xz ;  C := 0.0 ;

... Summations  of  83*L  terms  X.. :
For  j = 1 to 83  do {  for  k = 1 to L  do {
    X := Xz - E - Xy ;  Xz := -Xy ;  Xy := X ;
    S1 := S1 + X ;                ...  (True sum  S) - 1 .
    SS := SS + X ;                ...  Simple sum  SS .
    Y := C + X ;  T := SC ;
    SC := SC + Y ;                ...  Compensated sum  SC .
    C := (T - SC) + Y  }} ;       ...  Don't remove parentheses!

... Display results and their errors measured in  ulps :
ES := (((SS-H)-H) - S1)/E ;  EC := (((SC-H)-H) - S1)/E ;
Display( "True sum     S = ", H+S1+H );
Display( "Simple sum  SS = ",SS,"  in error by ",ES," ulps");
Display( "Compensated SC = ",SC,"  in error by ",EC," ulps");
Display( "(  One ulp  is an acceptable error in  SC .)" );
Stop ;  End CMPSUM.
```

Table 5 : Results from Program CMPSUM on various machines

Keyboard Input: L = 1000000 (83000001 terms summed)

Computer Family	Roundoff level (1 ulp) E	Ulps error Ex in sums Sx: Simple Sum ES in SS	Compensated EC in SC
IBM '370, 3090	1.39 E-17	-75000000	0
DEC VAX D format	1.39 E-17	27666666	0
G format	1.11 E-16	27666666	0
IEEE 754 Double	1.11 E-16	27666666	0
H-P 71B 12 sig. dec.	1.00 E-12	27666666	0
CRAY X-MP, Y-MP	3.55 E-15	27666667	*-27666664*
CRAY 2	3.55 E-15	-27666667	*27666666*
CRAY Double (all)	1.26 E-29	27666667	*-27666664*

These results are atypical in two respects. First, IEEE standard machines (H-P 71Bs conform to IEEE 854) have statistically unbiased roundoff, so simple summation rarely accumulates so much of it and compensated summation rarely improves sums so strikingly as here. Second, compensated summation fails utterly on CRAYs only for contrived examples. Compensated summation errs typically by a few ulps on CRAYs versus none or one on the others, never much more than simple summation, and is usually worth a try.

Software-Simulated Doubled Precision Retarded

Far more serious, but too complicated to discuss at length here, is how a missing guard digit retards software-simulated DOUBLE PRECISION invokable from most of CRAY's Fortran compilers. It slows programs reportedly by factors from 25 to 45 compared with ordinary REAL floating-point. There are two reasons for this.

First, CRAY's simulation aims to be faithful to the style of its REAL arithmetic on X-MPs and Y-MPs, omitting the guard bit just as they do, but with 96 sig. bits instead of 48. Software has to be long, slow and fussy to do this.

Second, CRAY cannot use portable and faster techniques for rough simulation of DOUBLE PRECISION because they fail on CRAYs in the same way as compensated summation fails, but much more often. The techniques in question are summarized by S. Linnainmaa in "Software for Doubled-Precision Floating-Point Computations" *ACM Trans. on Math. Software* 7 (1981) 272-83. They would run at least twice and more likely thrice as fast as CRAY's simulation if they could be used safely. Which brings up a horrible thought:

> Programs that use compensated summation, or similar methods described by Linnainmaa, to gain extra accuracy contain no tell-tale signs to warn the uncomprehending reader that they work on all machines in Table 2 but fail on CRAYs.

What would happen if software, highly regarded elsewhere because it benefits from those methods, were imported by well-intentioned users to CRAYs? Casual testing is unlikely to expose the risks.

The thought is too horrible.

CRAY's MULTIPLICATION AND DIVISION

CRAY's floating-point multiplication is fast but somewhat ragged; and since CRAY's floating-point division Y/X is composed from a product (1/X)*Y , division is ragged too. Unlike almost all other computers, whose products and quotients are always either correctly rounded or chopped, CRAYs suffer from bigger rounding errors. Their source is CRAY's abbreviated multiplier hardware, which saves a little time and space by dispensing with almost 36% of the bits that other conventional multipliers produce internally in the course of their operation. The abbreviation is now done in a commutative way (X*Y = Y*X) though long ago this was not so.

Different CRAYs attempt to compensate for that abbreviation in different ways. Some have an " **F* " multiply instruction that tends to under-compensate; some have an " **R* " instruction that tends to over-compensate; some have both, and then the compiler chooses them mostly in accordance with programmers' directives.

Division Y/X entails usually 4 instructions on a CRAY ;

> estimate a reciprocal R ≐ 1/X to 30 sig. bits,

> obtain a correction factor C = 2 - R*X ≥ 1 , and then

>> compute Y*R*C in lieu of Y/X .

The last product can be computed as Y*(R*C) , which tends to be more reliable, or as (Y*R)*C , which can be faster when Y*R and C can be computed in parallel. The last two multiplies can be **F*s or **R*s. Also uncertain is the approximate reciprocal R , which can vary very slightly from one CRAY model to another.

Since different CRAYs and different compilers have been known to get different results from the same Fortran program and input data, generalizations about CRAY's rounding error must cover a spread like the entry after "CRAY" in Table 3.

We shall describe the error in *ulp*s (*u*nits in the *l*ast *p*lace). **F* multiplication is known to err often by over 1 ulp but never by more than 1.23 ulps. **R* multiplication errs by under 0.83 ulp. CRAY's division errs over twice as much as multiplication; I do not yet know division's worst case errors. (At this CUG meeting, CRAY's Jim Kiernan reported finding errors almost as big as 2.5 ulps by random testing.) A complication peculiar to CRAYs is that their errors in expressions X*Y and Y/X depend upon both X and Y rather than just the ideal values of X*Y and Y/X respectively, as is the case for machines in Table 2, none of whose operations err by more than 0.5 ulp if rounded nor as much as 1 ulp if chopped.

Measured in ulps, CRAY's errors exceed what might be expected from any other machine by so little as appears not to matter. In fact the excess is more than enough to derail numerous programs that have been *proved*, both in practice and in theory, to work reliably on *all* other commercially significant machines. There is more to roundoff than its magnitude. Rounding errors are not random but correlated by mathematical rules upon which successful programs sometimes depend for their success, and upon which many a programmer depends because his mental model of arithmetic, based perhaps upon experience with calculators, takes those rules for granted.

To describe the effects of roundoff, we shall let brackets [...] distinguish a computed value from an ideal value of an expression. For instance, we shall write [Y/X] for the value [Y*[R*C]] a CRAY computes in lieu of Y/X . Only CRAYs can get [X/X] ≠ 1 for nonzero finite X ; **F* multiplications yield [X/X] < 1 at half of randomly chosen values X ; an **F* and then **R* yield [X/X] < 1 at about one X in 6 but [X/X] > 1 at one in 33 . Only on CRAYs *must* [21.0/3.0] be a non-integer; with two **R*s it is 7.000000 000000 03553 , otherwise 6.999999 999999 96447 .

What comes to mind is a *signature*, a way for a program to learn at run-time whether it is being executed on a CRAY regardless of its serial number or its compiler. Compute both expressions

(62.0*63.0)/63.0 and (63.0*63.0)/63.0 ,

and if *neither* is an integer then the computer is probably a CRAY or infringing CRAY's patents. (Thanks to Alex Liu.)

AMOD Misbehaves

When X and Y are positive, the Fortran remainder function

AMOD(Y, X) := Y - X*(integer part of Y/X)

is expected to satisfy 0 ≤ AMOD(Y,X) < X . CRAY's AMOD used to violate this constraint frequently; whether CRAY's current (and slower) AMOD always conforms to expectations is not yet known.

Sethian gets into Trouble

Prof. J.A. Sethian wished to compute one of the angles in a right triangle with sides X and Y not both zero. Two mathematically equivalent but computationally different formulas were considered:

Angle = PI/2 - ATAN(Y/ABS(X)) , and
Angle = ACOS(Y/SQRT(Y*Y + X*X)) .

The first formula malfunctioned at X = 0.0 on machines with no *Infinity*, and inserting a test and branch for 0.0 seemed so high a price to pay for such a rare event that Sethian opted for the second formula, which appeared robustly portable. But when he ran his program on a CRAY it sent him an obscure message that he decoded, after a week's help from local CRAY engineers, as

" *Invalid* ACOS(Argument > 1)."

Experiments with both **F* and **R* now reveal that [Y/[SQRT[Y*Y]]] exceeds 1.0 at roughly 7% of randomly chosen arguments Y on CRAYs. Before condemning Sethian's naivety in neglecting to first test the validity of ACOS's argument, reflect on this:

On all commercially significant machines *except CRAYs*,

-1.0 ≤ [Y/[SQRT[[X*X] + [Y*Y]]]] ≤ 1.0

is provably valid despite five rounding errors, though the proofs for binary, for hexadecimal (IBM '370), and for decimal (calculator) arithmetics do differ.

So testing the argument of ACOS is necessary only on CRAYs.

Monotonicity Fails
The worst aspect of CRAY's arithmetic is our uncertainty about its properties, rather than its specific ulp-bounds. Consider *Monotonicity*, for example. With conventional arithmetics, if $A*X \geq B*Y$ in exact arithmetic, then the computed values must satisfy $[A*X] \geq [B*Y]$, and if $A/B \geq Y/X$ then $[A/B] \geq [Y/X]$, despite roundoff. CRAY's arithmetic is not monotonic this way, with consequences that are worse than annoying.

A Quadratic Equation becomes Unreal
Suppose that the coefficients A, B, C of a quadratic equation $Az^2 - 2Bz + C = 0$ have been derived in such a way as assures physically meaningful real roots; that means that coefficients in memory would satisfy $B^2 \geq A*C$ if this could be computed exactly. The expression SQRT(B*B - A*C) will be encountered in the course of computing the desired roots, but it must be computed with four rounding errors. On conventional computers, $[B*B] \geq [A*C]$, so no precaution need be taken to avoid stopping with a message like

" *Invalid* SQRT(Argument < 0) ".

But precautions are mandatory on CRAYs. For instance, if

A := 33554432*16777216 = 2^{49} = $4032\ 800000\ 000000_H$,
B := 16777215*16777217 = $2^{48} - 1$ = $4030\ FFFFFF\ FFFFFF_H$ and
C := 10610063*13264529 = $2^{47} - 1$ = $402F\ FFFFFF\ FFFFFE_H$, then
D := B*B - A*C = +1 in exact arithmetic.

Other computers' D = [[B*B] - [A*C]] ≥ 0 ; CRAYs' D = -2^{48} when **F* multiplication is used. If **R* is used, try

T := 10610063*13264529 + 1 = 2^{47} = $4030\ 800000\ 000000_H$,
A := 2*78399859 + T = $4030\ 800009\ 5892E6_H$,
B := 3386665*58769639 = $4030\ B504FF\ FFFFFF_H$ and
C := 2*(A - 9971313) = $4031\ 800008\ C06C75_H$. Then
D := B*B - A*C = 1550789 in exact arithmetic.

Other computers' D = [[B*B] - [A*C]] ≥ 0 ; CRAY's D = -2^{48} .
Therefore on CRAYs something slightly more complicated than the expression SQRT(B*B - A*C) must be used even though this one malfunctions so rarely that instances of failure are practically impossible to find by the usual methods for testing programs.

Imagine now that this simple expression appears in a program with an impeccable reputation on IBM 3090s, DEC VAXes, Suns, etc., and with a published proof of correctness as part of its pedigree. And suppose the program is imported to a CRAY by a conscientious user who first runs tens of millions of test cases before passing the program around among his friends, who incorporate it deep in the bowels of their own work. Many months pass; then something strange happens. Whose fault is it? Who will have to repair it?

Simple Specifications rendered Complicated
Suppose a program must be written to compute a continuous function Z = Z(X,Y) according to the following specifications:

S1: Floating-point input data X and Y will be non-negative; and $X \leq Y$ more often than not.
S2: If X > Y then Z := X*arctanh(Y/X) + Y*ln(arccos(Y/X))
else Z := X*ln(2) .

§3: The program should invoke, from the run-time Math. library, high *relative* accuracy implementations of the functions
$\text{arctanh}(Q) := \ln((1+Q)/(1-Q))/2$ and
$\arccos(Q) := 2\arctan(\text{sqrt}((1-Q)/(1+Q)))$
that may well use machine-dependent formulas, different from those shown here, to ensure accuracy for all Q .

§4: Division may be slow; do not waste time computing quotients $Q := Y/X$ that will not be used when $X \le Y$.

§5: The program must be portable to a wide variety of machines, among them some on which *Infinity* is unavailable for arctanh(1) , ln(0) or Y/0 , so avoid computing these.

Why can't the program consist of just the short statement in §2 ? That program works correctly on practically all computers with built-in division hardware because they always compute rounded or chopped quotients $Q = [Y/X] < 1$ whenever $X > Y \ge 0$. The proof is easy: call the computer's number next less than 1.0
$U := \text{NEXT1}(0.5) = 1.000...000 - 0.000...0001$;
evidently $Y \le U*X$ in exact arithmetic, so $Y/X \le U$ and hence monotonicity would imply $[Y/X] \le U < 1$, as claimed.

CRAYs are the exceptions; occasionally they compute $[Y/X] = 1$ when $X > Y > 0$, and whether they can possibly get $[Y/X] > 1$ is not yet known. Therefore a short program derived too directly from §2 will occasionally attempt to compute arctanh(1) and ln(0) on CRAYs, and may for all we know be capable of asserting
" *Invalid* ARCCOS(Argument > 1) ".
Other machines are provably incapable of such misbehavior.

How would you program a portable and accurate computation of Z ? (Do not let relative accuracy evaporate when Z is near 0 .) Will your program be portable to CRAY X-MPs, Y-MPs and 2's ? How long will you spend on this task? Who will pay you for it?

Paranoid Programmers

Programmers cope with the uncertainties of CRAY's arithmetic by introducing extra *defensive* (not to say *paranoid*) tests and branches to guard against rare events that never happen on other machines. These expedients inflate the cost of developing, using and maintaining numerical software for Science and Engineering:

- Tests involve roundoff-related thresholds that take programmers extra time first to determine, and later to change.
- Tests and branches increase the programs' capture cross-section for programming errors which first prolong debugging and then undermine confidence that debugging is complete.
- Tests against rare data-dependent events erode programs' speed.

Worst of all for CRAY users, numerical software first developed and *proved* reliable on other machines cannot be used confidently on CRAYs without prior scrutiny for obscure malfunctions. These malfunctions occur far too rarely (perhaps only for " contrived data ") to be exposed by casual testing, yet not rarely enough to be ignored.

Accuracy (Eps) vs. *Precision*

On computers other than CRAYs, the accuracy of each arithmetic operation can be inferred from the precision to which results are stored; each operation is accurate to 1/2 ulp if rounded the way better binary and decimal calculator arithmetics do it, 1 ulp if chopped the way IBM '370 hexadecimal arithmetic does it. On CRAYs, arithmetic operations can err by rather more than that, as we have seen and recorded in Table 3. Consequently, software that depends upon relations between accuracy and precision, or presumes that addition, subtraction, multiplication and division are all about equally accurate, can malfunction on CRAYs.

Gauging the precision of numbers in memory is a comparatively easy experiment to perform at run-time, and many published ostensibly portable programs perform it more or less reliably. One of them is the function subprogram NEXT1(X) supplied along with RATAREA above. On practically any computer, CRAYs and all those listed in Table 2 included, the floating-point numbers adjacent to 1.0 are NEXT1(0.5) and NEXT1(2.0) . Hence the relative precisions of numbers in memory, measured by the half-widths of gaps that separate them, lie between

```
       PrecUnder1 := 0.5*((0.5 - NEXT1(0.5)) + 0.5)    and
       PrecOver1  := 0.5*(NEXT1(2.0) - 1.0) .
```

The ratio PrecOver1/PrecUnder1 is the computer's floating-point *radix*, 2 for binary, 10 for decimal, 16 for hexadecimal arithmetic. Precision (but not Accuracy) is related to the number of significant *radix*-digits stored thus:

$$2.0*\text{PrecUnder1} = (\textit{radix})^{-(\text{No. of Sig. Digits of Precision Stored})}.$$

See the second and third columns of Table 3 for examples.

Accuracy can be no better than Precision, and may be worse if arithmetic is not rounded in the best way. On most computers the accuracy of arithmetic depends solely upon whether it is rounded or chopped and can be determined in an ostensibly portable way at run-time from one experiment with an arithmetic operation, say multiplication, thus:

```
    ... Compute bound  Eps  for relative error due to roundoff:
        E := 2.0*PrecOver1 ;
        P := (1.0+E)*(1.0-E) ; ... = [1 - E²]  rounded or chopped.
        Eps := if  (0.5-P) - 0.5 = 0.0  then  0.5*E  else  E .
```

Computed this way on any machine listed in Table 2, Eps is a tight upper bound for the relative error committed by any rational floating-point operation. Eps works because all those machines either round correctly or chop the way IBM '370s do. Alas, nothing so simple works for CRAYs.

On CRAYs, different operations have different error bounds whose estimation in a machine-independent way would require an elaborate test suite far more devious than the simple computation above. No applications programmer should be expected to embed such a suite, not even if he knew it was needed, in a would-be portable program that depends upon Eps .

The Importance of the Relative Rounding Error Bound Eps
How might numerical software depend upon Eps ? Portable programs that compute special functions from their infinite series can use Eps to decide how many terms are necessary to achieve results as accurate as the machine deserves. A portable program that solves an equation by iteration can use Eps to decide when to quit iterating rather than waste time dithering over a solution's last few digits that are probably obscured by roundoff anyway.

Why compute Eps at run-time? Why not let a user insert Eps as a constant into the program? That seems more efficient. But few users know their hardware well enough to choose Eps correctly; and fewer read all the programs they use closely enough to learn when Eps is needed and where. Much valuable numerical software is distributed in source form over a network of diverse computers, often to be recompiled and used without sentient scrutiny unless warnings appear at compile-time or anomalies appear at run-time.

Why not let compilers be responsible for Eps ? This is the gist of what are called *Environmental Inquiries* that have begun to appear in standardized programming languages. Alas, language standardizers seem to think that roundoff is characterized by the precision of numbers in storage. They confuse Eps with smaller numbers PrecUnder1 and PrecOver1 that we derived above directly from NEXT1(X) , so what they call Eps tends to underestimate the effects of roundoff on CRAYs.

Stopping Criteria Confounded
Secant Iteration $x_{n+1} := x_n - f(x_n)(x_n-x_{n-1})/(f(x_n)-f(x_{n-1}))$ is one of many ways to solve an equation $f(x) = 0$ numerically. When should iteration be stopped? One stopping criterion waits until the iterates x_n have settled down in some sense; but they may never settle down if the computed values of $f(x)$ are too badly obscured by roundoff. Even if some program logic forces the iteration ultimately to stop, many iterations can be wasted on futile attempts to compute the desired solution more accurately than roundoff in $f(x)$ allows.

Better stopping criteria take account also of whatever is known about uncertainty in $f(x)$. If computable at a tolerable price, a bound $E(x)$ for error in $f(x)$ due to roundoff can serve to stop iteration whenever the computed value of $f(x_n)$ is no bigger in magnitude than $E(x_n)$. Further details applicable when $f(x)$ is a polynomial appear in "A Stopping Criterion for Polynomial Root Finding" by D.A. Adams (1967) *Comm. A.C.M.* **10**, 655-8. Here the partial fraction expansion of a rational function $f(x)$ will be discussed instead, for reasons that will be clearer later.

Let $f(x) := b_1/(a_1-x) + b_2/(a_2-x) + \ldots + b_N/(a_N-x) + c$ where all b_n are nonzero and every a_n distinct. Here is a program that computes simultaneously $F \doteq f(x)$ and $E \doteq E(x)$ to bound the contribution of roundoff; i.e. $E(x) \ge |F - f(x)|$:

```
F := c ;  E := 0 ;
For  n = 1 to N  do  {
    Q := bn/(an - x) ; ...  most time is spent here.
    F := F + Q ;
    E := 2.0*|Q| + E + |F|  } ;
E := E*Eps .   ... uses the roundoff bound  Eps  above.
```

On practically all computers roundoff in F never exceeds E but often approaches it, so stopping iteration when $|F| \le E$ avoids wasteful dithering. The exception is a CRAY, on which this program's E is too small for two reasons. One is that Eps is too small when computed in the ostensibly machine-independent way described above; it should be roughly doubled to be valid for all CRAYs. More serious, the formula for E is invalid for machines like CRAYs that lack a guard digit in subtraction; to correct E the subexpression " 2.0*|Q| " has to be replaced by

" $(2.5 + (|a_n| + |x|)/|a_n - x|)*|Q|$ "

which is far more expensive and produces a bigger bound E hardly ever approached by roundoff. Without the corrections, roundoff's contribution to F can easily exceed E on CRAYs on which, in consequence, F can sometimes consist entirely of roundoff and still violate the condition $|F| \le E$. Subsequent iterations on a CRAY may well dither or, worse, jump away from the desired root only to stop after reaching some arbitrary limit imposed upon the number of iterations by a paranoid programmer.

In short, because a CRAY's accuracy is not correlated with its precision in the same way as on other machines, some risk exists that an ostensibly portable program which is provably reliable and efficient on other machines will run too long on a CRAY and then stop with an inaccurate answer. Such mishaps must be rare, but devils to debug. And the debugged program may be uneconomical to run on a CRAY or on anything else. Another such case follows.

An Apology to Language Implementors

This is a good way to fill an otherwise blank quarter page.

Exasperated, I have at times railed at well-intentioned but ill-informed compiler writers for hurting engineering and scientific computation as much as computer arithmetics designed with no thought for anything but speed. However, compiler writers have to cope with those arithmetics too. At times they try to reconcile the irreconcilable: ambiguous comparisons, dubious environmental parameters like Eps, a tempting "optimization" that always runs faster but perhaps on very rare occasions less accurately or even wrong, ...

They deserve our sympathy and sometimes forgiveness, and better advice than they usually get. I commend to their attention a paper "Compiler Support for Floating-point Computation" by Charles Farnum in *SOFTWARE - PRACTICE AND EXPERIENCE* **18** (1988) 701-9.

ANOTHER PROGRAM WE CAN'T HAVE

CRAYs run most matrix computations faster than almost every other machine, but there are exceptions. One of the newest programs, one that runs faster on every computer than all competing methods, runs inaccurately only on CRAYs. Therefore its distribution to a larger community of computers that can run it well is in jeopardy.

Parallel computation of all eigenvalues and eigenvectors of real symmetric matrices has challenged numerical analysts for more than a decade. Until recently, there seemed to be no way to guarantee the orthogonality of eigenvectors computed on multi-processors in parallel without extensive communications. Trouble arises because computed eigenvectors can be rotated by angles of the order of

roundoff/(difference between eigenvalues) ,

so near-coincident eigenvalues have eigenvectors we dared not try to compute concurrently on independent processors each ignorant of the effects of the others' rounding errors.

To defeat this trouble, we have learned which rounding errors most affect the eigenvectors, and how to correlate roundoff on different but identical processors so that orthogonality of all eigenvectors can once again be guaranteed. The most recent report on this subject is "On the Orthogonality of Eigenvectors Computed by Divide-and Conquer Techniques" (May 3, 1990) by D. C. Sorenson and P. T. Peter Tang, the first at Rice University, Houston TX 77251-1829, the second at Argonne National Lab., Argonne IL 60439-4801. In their *Concluding Remarks* they say

" Finally, it is unfortunate that neither Reform nor Acc_Sec *(the two new methods they compare)* would work on machines whose arithmetic subtraction lacks a guard digit (or bit); notable examples are Crays and CDCs. It is even more unfortunate because, despite the many anomalies those arithmetics offer, merely adding a guard digit (or bit) would allow Acc_Sec *(their best method)* to work. The ultimate misfortune is that robust programs such as Acc_Sec may never appear in a federally funded software library such as LAPACK simply because the program may not run on a few aberrant machines whose manufacturers have not implemented the extra guard digit in their hardware."

Their assessment cannot be faulted. I shall try to explain the main political and numerical facts underlying their assessment so that readers of this document can better judge its significance.

Federal subsidies for the development of numerical software derive in part from a Congressional inclination to foster American supercomputing. Subsidies are as indispensible here as for public transportation and the interstate highway system. Administrators of some research grants have interpreted the mandate of Congress to imply that *all* subsidized software must be portable to *all* American supercomputers, CRAYs among them. Unfortunately that policy hurts the larger community of computers listed in Table 2 by precluding support for software that exploits good arithmetic properties they all possess but CRAYs do not. That policy is so unwise that it has to change, and indications are that it will.

Why do the new eigenvector algorithms fail only on CRAYs? How is failure traceable to subtraction's lack of a guard bit? The task

is to compute accurately the differences between the roots of the equation $f(x) = 0$ and the coefficients a_n that appear above in a partial fraction expansion for $f(x)$. Those differences serve as divisors in the formulas for eigenvectors. Since all $b_n > 0$, the roots in question alternate with the a_n and can come awfully close when some b_n are tiny, so the displacement from each root to its nearest a_n is the variable manipulated instead of x . This entails the computation of differences $a_m - a_n$; these have to be mutually consistent especially when small, which is where lack of a guard bit first causes trouble. But worse is to come.

If data and results are declared REAL*8 , a small but critical part of the computation has to be performed to roughly REAL*16 accuracy. Most computers, like CRAYs, must simulate REAL*16 in software; most compilers, unlike CRAY's, offer no support for REAL*16, so simulation has to be performed using exclusively REAL*8 variables. Many ostensibly portable algorithms simulating arithmetic to roughly twice the accuracy of the hardware have been published over the past two decades; all but one fail on a CRAY for lack of a guard bit. The exception, based upon a suggestion in my paper "A Survey of Error Analysis" in the proceedings of the IFIP Congress of 1971 in Ljubljana, runs so much slower than the others that it is uncompetitive. That is why Sorensen and Tang chose for Acc_Sec a REAL*16 simulation that fails on CRAYs though it is provably fine for all machines in Table 2.

An obvious remedy comes to mind: distribute two versions of Acc_Sec, one for CRAYs and the other for everyone else. The CRAY version can use REAL*16 explicitly where the other just simulates it. Aside from its nuisance and expense and politics (how many manufacturesrs are entitled to their own versions of software developed at public expense?), this remedy has a flaw:

Because CRAY's REAL*16 software is so slow, CRAY's version of Acc_Sec would run substantially slower than the ostensibly portable version runs on a CRAY. After a while some innocent user, whose CRAY is just one of many different machines on a network, will transfer a major applications code, containing the ostensibly portable version of Acc_Sec buried deeply inside it, to his CRAY and, after casual testing that excludes close scrutiny of the source code, will conclude that it runs faster and scarcely less accurately than all competing software. A calamity is just a matter of time; who will be blamed for it?

To defend against calamity, the program could try something like

```
    If  (1.0 - NEXT1(0.5)) ≠ ((0.5-NEXT1(0.5)) + 0.5)  then  {
        Protest("This computer lacks a guard bit for  + and - .");
        Crash and Burn } ;
```

which usually detects the lack of a guard bit unless compile-time "optimization" is over-zealous. Is the remaining risk tolerable?

MUST EXCEPTIONS BE FATAL ERRORS ?

Exceptions arise in situations like (...)/0.0 , SQRT(-3.0) , (10000.0)**617 (overflows), (0.0001)**617 (underflows) , and many others. What makes them exceptional is not that they are errors in the sense of *Sin* but that the computing ambience may be unable to cope with them in ways that serve every legitimate interest. In short,

Only Exceptions handled badly are Errors.

Some readers will wish to debate this motion. I wish to persuade readers that CRAYs do handle exceptions badly, and that better possibilities are feasible and worth-while. Two pregnant examples will be presented; a thorough discussion must be deferred.

Let H := 0.5 and U := H + H ; so U = 1.0 , but don't tell it to the compiler. Can you find positive floating-point numbers C for which

```
((U*C)*H)*H        overflows,   but a bigger expression
((H*C)*U + C)*H    does not?
```

Only on a CRAY . In unoptimized Fortran, try C := D + D for

```
4.01*(2.0**8187)  <  D  <  5.32*(2.0**8187) .
```

That will do it. This suggests that CRAY's overflow threshold is ambiguous, obscuring the answers to questions like ...

-> If my program malfunctioned because of overflow,
 how big must some intermediate result have become?
-> What data can cause overflow in my program?
-> How wide a range of data should my program have to handle?

In general, CRAYs treat overflow fuzzily, as if numbers that nearly overflow were believed to be invariably symptomatic of a computation that will have to stop sooner or later anyway either because the program's design is mistaken or because input data has been chosen in bad taste.

A false belief has produced a bad policy. Situations do arise in which, for reasonable data with reasonable results, a reasonable program will founder in unavoidable overflows or other exceptions unless it is encumbered unreasonably with precautionary tests that could almost all be eliminated if computers handled exceptions in a reasonable way. The example below was drawn by J. W. Demmel from the LAPACK project.

To compute an eigenvector **v** of a matrix B after its eigenvalue b has been located, one must solve $(B - bI)\mathbf{v} = \mathbf{y}$ for **v** with a right-hand side **y** that is not critical so long as it is very tiny (**o** would be nice) while **v** is not. This is feasible because $(B - bI)$ must be singular or nearly so; that is what " b is an eigenvalue of B " means.

The solution process begins with Gaussian elimination tantamount to factorizing $(B - bI) = \mathbf{PLU}$ where **P** is a permutation matrix induced by pivotal exchanges that conserve numerical stability, **L** is a lower triangular matrix with 1 's on its diagonal and no bigger magnitudes below, and **U** is upper triangular with the pivots on its diagonal. Since $\det(\mathbf{P}) = \pm 1$ and $\det(\mathbf{L}) = 1$, it will come as no surprise that **U** must be singular or nearly so,

just as $(B - bI)$ is. In fact, the better b approximates an eigenvalue, the more nearly singular **U** tends to be. Indeed, should **U** be exactly singular because a diagonal element has vanished, the following process will simplify; but luck that good is rare and can be ignored here.

All that remains is to perform, say, *back-substitution* to solve $\mathbf{Uv} = \mathbf{w}$ for the eigenvector **v** with any right-hand side $\mathbf{w} = \mathbf{PLy}$ that is very tiny while **v** is not. Typically, **w** can consist of random numbers comparable in size with the rounding error threshold r, though normally we do better than that, and then **v** consists typically of numbers of moderate magnitudes. Finally, **v** is divided by its biggest element to normalize it.

The foregoing process can fail because of intermediate overflows even though none of **B**, b or **v** is extraordinary. Such failures are extremely rare; on a CRAY they may never happen unless **B**'s dimension exceeds a few hundred. The simplest failure mode arises when each element of **v** except the first is far tinier than the element before; since the elements of **v** appear in reverse order they can grow past the overflow threshold before the program has a chance to divide by the biggest.

The remedy is clear; whenever overflow occurs, multiply all the elements of **v** and **w** computed so far by a tiny number, perhaps r^4, and resume computation at the element that overflowed. Even though the overflow test resides outside the innermost loop, this turns out to be awkward if not impossible to program in CRAY's Fortran without severely retarding the speed of normal cases.

How can the program discover whether overflow occurred and then branch to the correct place without forestalling the scheduling of concurrent operations? The danger to avoid is to ask whether an unfinished vectorized operation has overflowed, and be told it hasn't when it will. The nuisance to avoid is waiting for that operation to complete before starting anything else. The hardware seems not to preclude a program that steers between these hazards, but currently it probably cannot be written purely in Fortran.

To begin with, CRAY 2s seem to handle exceptions differently than other CRAYs, though I am not sure how. Second, signals to and from the operating system are involved, something that is provided for in C but not in Fortran except via C . Then there are dangerous interactions with compiler optimizations that have to be prevented in clumsy ways because CRAY Fortrans know nothing about C's signals.

LAPACK has abandoned this problem, making no attempt to recover from overflow nor to forestall it (which would waste too much time on tests and branches). Users will have to take their chances which, considering the Risk Equation, can only worsen with the passage of time.

The fault does not lie exclusively with CRAY. Humane handling of exceptions is not a high priority in a computing industry which has yet to agree upon their names. CRAY is merely somewhat worse than the current norm. (Apple is currently best!)

CRAY, however, is very visible to the industry. Because CRAY's exception handling is so primitive, it sets a bad example for all others to follow. For instance the Inmos T800 Transputer claims to conform to IEEE 754 although it has elided three flags (one for Invalid Operations like 0.0/0.0 , one for Overflows like 1000.0**(1000**2) that produce inexact infinities, and one for Divisions-by-Zero like 3.0/0.0 that produce exact infinities) into one flag just as CRAY has done. Hence a continued fraction that signals a division-by-zero, whose infinity later turns harmlessly into 0.0 by division, is indistinguishable from more dangerous indeterminate expressions like 0.0/0.0 that turn into a *NaN* (*N*ot-*a*-*N*umber), unless the machine is trapped and run back over potentially offending operations in a kind of slow mode.

Another bad example is the Intel 860, which handles exceptions in a different way about as primitive as CRAY's. Also the i860 does division in software letting users choose between very slow but correctly rounded division or moderately slow division about as bad as CRAY's. The problems the i860 presents to compiler writers boggle my mind but I can still see CRAY's influence.

In the absence of thoughtful leadership, the industry will drift into a situation where everyone's exception handling is primitive and different, and automatic exception handling in portable software is commercially impossible. Users who have to face data that precipitates exceptions will be presumed to have been served by software so hardware-dependent and so slow that it will not yet have been written.

RECOMMENDATIONS FOR CHANGE

The first recommendation, and an urgent one, is that a guard bit be appended to CRAY's add/subtract hardware, which would then operate thus:

Steps in Guarded Subtraction D := A - B :
~~~~~~

§1: Swap if necessary to get |A| ≥ |B| .
§2: Shift B's sig. bits right enough to equalize exponents.
§3: On a CRAY 2, round off B's sig. bits past A's 49th.
On a CRAY ?-MP, discard B's sig. bits past A's 49th.
§4: Subtract what is left of B from A .
§5: Shift left or right to normalize the difference.
§6: Discard any bit past the result's 48th.

The change consists in numbers " 49 " that used to be " 48 ". Its impact upon CRAY's existing *correct* codes will likely be imperceptible or else almost certainly improve them a little.

Further, this change should be retrofitted to all current CRAYs. Otherwise, since software for CRAYs is probably expected to run (after recompilation) on *all* CRAYs, no market would exist for new software that exploited improved arithmetic, so nothing much would change after all. Keeping all CRAYs no more diverse than they are already helps spread the cost of promulgating numerical software over the widest possible market.
~~~~~~

Objectors to retrofitting may cite the substantial costs entailed by any change whatever. The revalidation of certified codes comes to mind at once. However, since continual change has been going on to correct bugs in hardware and (more often) compilers, and since the CRAY 2's floating-point arithmetic already differs from the others by more than enough to jeopardize validations that were conducted in ignorance of the difference, the costs of the change I propose seem tolerable considering the actual cost now of computing on CRAYs. Besides, proper validation should include more than a battery of test cases; error analysis should be part of validation too, and that part is eased by the proposed change.

One way to compensate for the costs of change would be to offer a faster alternative to the present DOUBLE PRECISION arithmetic simulation package and library. The faster alternative would be about three times as fast, but not so accurate as the present package could become under the influence of the new guard bit.

The second recommendation could cost an order of magnitude more to implement than the first, and would probably deliver an order of magnitude less value; but it is still worth-while:

Repair multiplication and division.

The multiplier array should be expanded to produce more nearly a correctly rounded result. This would complicate the multiplexer that chooses whether to normalize the final product by one bit, since rounding would be included. The present round-before-shift is unsatisfactory. Only **R* multiplication is specified here; **F* can do whatever comes naturally.

Having repaired multiplication we can address division. As it is now, division Y/X starts with a rough reciprocal $R \doteq 1/X$ that is biased low; so $C = 2 - R*X \geq 1$. This gratuitously loses a sig. bit. With an improved multiplier, it becomes worth-while to bias R slightly high, and then get $C \leq 1$. Then the compiler should use **R* multiplications, instead of one **R* and one **F* , to get a better quotient Y*(R*C) .

The third recommendation is that exception handling be improved, in both hardware and software, to an extent that will not likely affect existing Fortran codes adversely after recompilation, but should offer considerably improved functionality. Lacking adequate knowledge of the present state of affairs, a wish-list is all I can offer.

I wish for a return to the CDC 6600's Infinity and Indefinite, but more along the lines of the IEEE standards' Infinity and NaN . These should be correlated with amendments to compilers that would ensure correct comparisons involving those symbols. The possible utility of a signed zero, valuable for conformal maps of slitted domains in fluid flow calculations, would have to be balanced against its possible disruption of comparisons.

The one flag that now catches all exceptions should become two to distinguish creation of Infinity from creation of a NaN, but eliding division-by-zero with overflow is probably acceptable. Finally, Retrospective Diagnostics should be provided; this amounts to a record of flag-raising exceptions compactly hashed by site in the program. Details some other time.

ACKNOWLEDGMENTS
Thanks are due to David H. Bailey of NASA Ames Research Center for good advice, and to Z-S. Alex Liu for running innumerable examples on diverse machines, including a CRAY X-MP. And quite a few CRAY employees have run my examples and tried to explain aspects of CRAY hardware and compilers. Financial support has come partly from the University of California, partly from the U. S. Office of Naval Research (# N00014-90-J-1372), and partly from the National Science Foundation (CCR-8812843). No computer maker has provided material compensation for this work.

. . . .

EPILOGUE: June 1991

An abbreviated version of the foregoing material was presented to the CRAY Users' Group (CUG) at their meeting 10 April 1990 in Toronto, Canada. Later in that meeting, after I had departed, the Group framed a petition to CRAY asking that ...

1) The accuracy of the math. library of transcendental functions be improved.

2) CRAY adopt IEEE Standard 754's arithmetic some time in the future, preferably soon.

3) CRAY retrofit the guard digits missing from add/subtract and multiply in its present hardware, as suggested above.

Subsequently CRAY did improve substantially the accuracy of its math. library, and announced that the format of IEEE 754 but not all of its functionality would be adopted in some future CRAY machines. However, CRAY declined to retrofit anything.