INFORMATION PROCESSING 71 – NORTH-HOLLAND PUBLISHING COMPANY (1972)

A SURVEY OF ERROR ANALYSIS

W. KAHAN
Computer Science Department, University of California, Berkeley, California 94720, USA

Rounding error is just one kind of error, and an easier kind to analyze than some others. Error and uncertainty in data is a more important kind, and not so easy to estimate nor analyze; here is where error analysts are currently busiest. The most refractory kind of error is attributable to flaws in the design of computer systems, both hardware and software, caused primarily by misconceptions about the other kinds of error. These flaws should not be blamed entirely upon those systems' designers, who must contend with arbitrary directives from on high and conflicting advice from their customers; "Who shall decide when doctors disagree?"

"A little neglect may breed mischief...
for want of a nail the shoe was lost;
for want of a shoe the horse was lost;
and for want of a horse the rider was lost."

from *Poor Richard's Almanac*
Benjamin Franklin

1. INTRODUCTION

A horse, a rider, a battle, a crown; that they all might be lost for want of a nail is plausible though unlikely. How likely is anything important to be lost because of a rounding error? Before we answer this question, we might consider the inhabitants of a far northern city who are persuaded that their harsh arctic climate is really very healthy because they see so few sick people walking their streets. Will our logic be any better than theirs?

There is a natural analogy between illness and numerical inaccuracy. Germs and rounding errors are small, numerous, and best combatted by sanitary precautions which, alas, are all too frequently neglected, not so much because of their intrinsic difficulty or expense as because of indifference or ignorance. When that neglect breeds mischief, the doctor is called. Now the analogy breaks down; germs are more persistent than rounding errors. Among the achievements of the past generation of error analysts is their capacity to deal with roundoff in a comparatively routine way that medical practitioners could only envy. Of the various kinds of errors that confront error analysts, rounding errors are among the easier kinds to deal with theoretically, so let us deal with them first.

2. EXAMPLE OF ROUNDOFF ANALYSIS

Here is an example, solving the quadratic equation

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

to illustrate the routine by which a mathematician may dispose of roundoff. This example has been chosen because its analysis is relatively short but otherwise typical of small algebraic problems. The first formula that comes to mind,

$$R_{\pm} = (B \pm \sqrt{B^2 - AC})/A\,,$$

is well known to be a poor way to compute the roots R_+ and R_- whenever one root is very much smaller in magnitude than the other; see fig. 1, which shows such a calculation done in 4-significant-decimal floating-point arithmetic.

Because the computed value of R_- is quite wrong, we might describe the computation as "unstable"; this is a correct conclusion from wrong reasoning, as we shall see. We might also be tempted to condemn the last subtraction for "losing" three significant decimals, though that subtraction has been

To solve $Ax^2 - 2Bx + C = 0$ with

$A = .1002,$ $B = 98.78,$ $C = 10.03;$

Use 4-significant-decimal rounded floating point arithmetic;

Set $D = B^2 - AC = 9757.4884 - 1.005006 \doteq 9757. - 1.005 \doteq 9756.;$

The roots are $R_{\pm} = (B \pm \sqrt{D})/A,$ where

$$\sqrt{D} \doteq \sqrt{9756.} = 98.772466... \doteq 98.77;$$

$R_+ \doteq (98.78 + 98.77)/A \doteq 197.6/.1002 = 1972.0559... \doteq 1972.$

$(R_+ = 1971.8055...);$

$R_- \doteq (98.78 - 98.77)/A = .01000/.1002 = .09980039 9... \doteq .09980$

$(R_- = .05076554...).$

Fig. 1. An unstable calculation.

```
C...    TO SOLVE A*X**2 - 2*B*X + C = 0.
        D = B**2 - A*C
        IF( D .LE. 0. ) GO TO 1

C...    REAL DISTINCT ROOTS RP AND RM WHEN D > 0.
        S = B + SIGN( SQRT( D ), B)
        RP = S/A
        RM = C/S
        GO TO ...

C...    COMPLEX OR  COINCIDENT ROOTS RR ± I*RI WHEN D ≤ 0.
      1 RR = B/A
        RI = SQRT( -D )/A
        ...
```

Fig. 2. A stable algorithm.

To solve $Ax^2 - 2Bx + C = 0$ with

$A = 47.51,$ $B = 47.45,$ $C = 47.39$

using 4-significant-decimal rounded floating-point arithmetic in the program of Figure 2;

Set $D = B^2 - AC = 2251.5025 - 2251.4989 \doteq 2252. - 2251. = 1.000;$

Set $S = B + \sqrt{D} \doteq 47.45 + 1.000 = 48.45;$ the roots are

$R_+ = S/A \doteq 1.0197853... \doteq 1.020$ $(R_+ = 1.000)$

and

$R_- = C/S \doteq .97812178 \doteq .9781$ $(R_- = .99747422...).$

Fig. 3. Poor results from a stable program!

Approximate solution of $ax^2 - 2bx + c = 0$;

$d = \{b^2(1+\mu_1) - ac(1+\mu_2)\}\{1+\sigma\}$;

When $d > 0$ estimate real distinct roots r_+ and r_-;

$s = \{|b| + (1+\rho)\sqrt{d}\}(1+\alpha)sgn(b)$; $(sgn(0) \equiv 1)$

$r_+ = (1+\delta_1)s/a$;

$r_- = (1+\delta_2)c/s$; go to ... .

When $d \leq 0$ estimate complex or coincident roots $r_r \pm ir_i$;

$r_r = (1+\delta_3)b/a$;

$r_i = (1+\delta_4)\{(1+\rho)\sqrt{-d}\}/a$.

Rounding errors: α for add, σ for subtract, μ for multiply, δ for divide, ρ for square root.

Fig. 4. Representation of rounding errors in fig. 2.

performed precisely and no more deserves condemnation than does any other bearer of ill tidings. The subtraction merely reveals an error half of which was committed at the beginning when $B^2 = 9757.4884$ was rounded to 9757.

A stable Fortran-like program to solve the quadratic is displayed in fig. 2; when applied to the coefficients A, B, C of fig. 1 it produced the roots correct to within one ulp. (An ulp is a Unit in the Last Place quoted.) Although in fig. 3 this program appears to lose half the figures carried, yet I insist that the program deserves to be called "stable"; the loss of figures could be charged against the data A, B, C if these coefficients were all uncertain by as much as ten ulps, for then they would specify an ill-conditioned problem whose solution is uncertain more because of its own data's uncertainty than because of my program's roundoff. To prove this, to exculpate my program, I submit the following analysis.

3. REPRESENTATION OF ROUNDOFF

The letters A, B, C,... are intended to be the names of real variables but the Fortran compiler interprets them as the names of cells in which are stored the values of real variables we shall call $a,b,c,...$ respectively. The variables A and a are not the same, though generally intended to approximate each other. A Fortran statement intended to compute, say, a quotient

$$R = S/A$$

causes instead the computation of, say,

$$r = (1+\delta)s/a$$

where the variable δ represents the contribution to r due to roundoff. For example, when $s/a = 1.019\,7853...$ is rounded to $r = 1.020$ then $\delta = (r - s/a)/(s/a) = 0.00021...$; usually the only information about δ that is used in an error analysis is an *a priori* bound; in this case the assertion

$$|\delta| < 0.0005$$

is valid independently of s and $a \neq 0$. More generally, to every arithmetic operation performed on a specific machine corresponds a data-independent bound which reflects the worst error that could possibly occur during that operation (in the absence of over/underflow). Customarily we assume that each floating point arithmetic operator # such as +, −, *, /, $\sqrt{}$, decimal-binary conversion, ... has, for every precision (word-length) pre-assigned to the cell called R, its own data-independent bound $\epsilon_\#$ for the relative error committed when the Fortran statement

$$R = S\#A$$

causes a new value, obtained by adjusting $s\#a$, to be stored in cell R. Whether the adjustment is by rounding or chopping is a minor issue to be discussed later; here rounding has been assumed. Whether $\epsilon_\#$ is a bound for $|(r - s\#a)/(s\#a)|$ or $|(r - s\#a)/r|$ is a matter of convenience for the an-

A slightly wrong solution to a slightly wrong problem.

Set $\tilde{a} \equiv a$, $\tilde{b} \equiv b$, $\tilde{c} \equiv c(1+\mu_2)/(1+\mu_1)$ *in the perturbed quadratic* $\tilde{a}x^2 - 2\tilde{b}x + \tilde{c} = 0$;

If $d \le 0$ *the roots* $\tilde{r}_r \pm i\tilde{r}_i$ *of the perturbed quadratic are approximated closely via the computed values;*

$$r_r = (1+\delta_3)\tilde{r}_r, \quad r_i = (1+\delta_4)(1+\rho)\sqrt{(1+\mu_1)(1+\sigma)}\,\tilde{r}_i .$$

If $d > 0$ *the roots* $\tilde{r}_\pm$ *of the perturbed quadratic are approximated closely by the computed values*

$$r_+ = (1+\theta)(1+\alpha)(1+\delta_1)\tilde{r}_+, \quad r_- = \tilde{r}_-(1+\delta_2)(1+\mu_1)/\{(1+\theta)(1+\alpha)(1+\mu_2)\}$$

where $\theta \equiv (s/(1+\alpha)\,sgn(b))/(|\tilde{b}| + \sqrt{\tilde{b}^2 - \tilde{a}\tilde{c}}) - 1$

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

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

Fig. 5. Assimilation of rounding errors in fig. 4.

alyst (and confusion for the student). Whether the customary assumptions can be validated for any particular computer system is one of the major issues to be discussed later. The arithmetic in fig. 3 is done in such a way that $\epsilon_{\#} = .0005$ holds for every operator. Finally, complicated Fortran statement like

```
D = B**2 - A*C
```

are interpreted as abbreviations for sequences of simpler statements like

```
T1 = B**2
T2 = A*C
D  = T1 - T2 .
```

The ways in which a Fortran compiler might introduce these invisible temporary variables is another major issue to be discussed later; here we assume each such variable to be, like all the others, restricted to 4 significant decimals.

Fig. 4 shows the relation between the program of fig. 2 and the values actually taken in storage by the variables $a, b, c, \ldots$. Each of the Greek letters in fig. 4 represents a rounding error about which we assume only that it is smaller in magnitude than $\epsilon = .0005$.

Let us assail this confusing profusion of Greek letters with the following question:

> Do there exist coefficients $\tilde{a}$, $\tilde{b}$, $\tilde{c}$, differing from a, b, c respectively by at most a few ulps, whose quadratic equation
>
> $$\tilde{a}x^2 - 2\tilde{b}x + \tilde{c} = 0$$
>
> has roots $\tilde{r}_\pm$ differing from the computed values $r_\pm$ respectively by at most a few ulps?

Yes, there are many such coefficients $\tilde{a}$, $\tilde{b}$, $\tilde{c}$; so many that a novice might have trouble finding any! One set is displayed in fig. 5, in which the coefficients' perturbation is confined to two rounding errors in c, while each root's perturbation amounts to five or fewer rounding; i.e., ignoring ϵ^2 terms,

$$|\tilde{c} - c| \lesssim 2\epsilon|c|, \quad |r - \tilde{r}| \lesssim 5\epsilon|\tilde{r}| \text{ for } r_+, r_-, r_r, r_i .$$

In effect, the program's first two rounding errors have been carried backward to c while the rest have been carried forward to the roots, We may compute

$$\tilde{c} = \frac{\{ac(1+\mu_2)\}b^2}{\{b^2(1+\mu_1)\}a} = \frac{2251}{2252}\,\frac{b^2}{a} = 47.3690322\ldots ,$$

which differs from the given value $c = 47.39$ by about 2 ulps, and then verify that the roots $\tilde{r}_+ = 1.01978\ldots$ and $\tilde{r}_- = .977691\ldots$ of the perturbed equation $ax^2 - 2bx + \tilde{c} = 0$ differ from fig. 3's

computed values $r_+ = 1.020$ and $r_- = .9781$ by less than 4 ulps.

4. A SLIGHTLY WRONG SOLUTION TO A SLIGHTLY WRONG PROBLEM

Do not be deceived by the last few computations, however small they make the errors seem to be. They do not say how close the computed roots $r_\pm$ are to the "true" roots of "the" quadratic equation; we have not yet identified the "true" roots because we have not yet identified "the" quadratic equation. Let that equation be

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

with coefficients A, B, C that are approximated by the variables a, b, c represented in storage. The values of A, B, C may be unknown but, if the calculation is worth doing at all, we must have bounds for them; for example, suppose the inequalities

$$|A-a|/|a| \leqslant 10\epsilon, \qquad |B-b|/|b| \leqslant 10\epsilon,$$

$$|C-c|/|c| \leqslant 10\epsilon$$

say all that is known about A, B, C. These inequalities imply that the true roots $R_\pm$ are uncertain by at least (actually much more than) a factor of about $(1 + 20\epsilon)$ because to two sets of coefficients satisfying the foregoing inequalities, say

$$A' = (1 + 10\epsilon)a, \qquad B' = b, \qquad C' = (1 - 10\epsilon)c,$$

$$A'' = (1 - 10\epsilon)a, \qquad B'' = b, \qquad C'' = (1 + 10\epsilon)c,$$

correspond roots satisfying

$$R'_+R'_- = C'/A' = \left\{\frac{1 - 10\epsilon}{1 + 10\epsilon}\right\}^2 R''_+R''_- ,$$

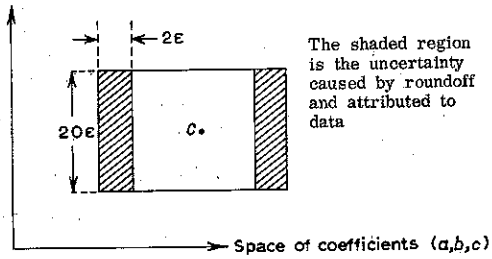
so either (R'_+/R''_+) or (R'_-/R''_-) differs from 1 at least as much as $(1 - 10\epsilon)/(1 + 10\epsilon)$ does. Compared with these relative uncertainties of 10ϵ in the coefficients A, B, C and consequently at least 20ϵ in the roots $R_\pm$, the additional relative uncertainties of 2ϵ in $\bar{c}$ and 5ϵ in $\bar{r}_\pm$ added by roundoff in fig. 2's program seem unobjectionable. See fig. 6.

Thus do we render the following verdict: The program in fig. 2 is not guilty of objectionable rounding error; the wrong answers in fig. 3 are scarcely more wrong than they deserve to be. But those answers remain wrong nonetheless! Is this progress?

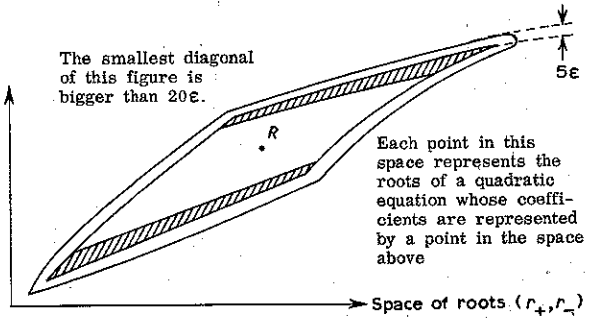
5. APPEAL TO PERTURBATION THEORY

We have made progress. Even if the intended coefficients A, B, C are *not* uncertain, but precisely equal to the stored values a, b, c,* the foregoing anal-

* This is assumed true for the remainder of this section.



The intended coefficients A, B, C are at the point C. The stored coefficients a, b, c are in the inner square. The perturbed coefficients a, b, c are in the outer rectangle.



The roots $R_\pm$ of the intended equation (A,B,C) are at the point R. The roots of the stored equation (a,b,c) are in the inner lozenge. The roots $r_\pm$ of the perturbed equation (a,b,c) are in the middle lozenge. The computed approximations $r_\pm$ are in the outer lozenge.

Fig. 6. Pictorial assimilation of rounding errors in figure 4. A slightly wrong solution to a slightly wrong problem.

ysis is helpful because it allows the error made during the computation to be summarized in a way that frees subsequent analysis from the messy details of the program and the computer's hardware. Here is the summary:

The computed "roots" r are close to actual roots $\tilde{r}$

(i.e., $|r-\tilde{r}|/|\tilde{r}| \lesssim 5\epsilon$)

of a perturbed quadratic equation $ax^2 - 2bx + \tilde{c} = 0$ whose coefficients a, b, $\tilde{c}$ are close to the given values a, b, c

(i.e., $|\tilde{c}-c|/|c| \lesssim 2\epsilon$).

All that is left of the program and its rounding errors is the pair of values $(5\epsilon, 2\epsilon)$ and the following question:

How much can the roots of a quadratic equation change when the last coefficient of the quadratic is changed by at most a little?

This question submits to conventional perturbation analyses. For example, we may regard each root R of $ax^2 - 2bx + c = 0$ as a function of c and compute the derivative

$$\frac{\partial R}{\partial c} = \frac{-1}{2(aR - b)},$$

whence the bound $|\Delta c| = |c-\tilde{c}| \leqslant 2\epsilon|c|$ implies that the error $|\Delta R| = |R-\tilde{r}|$ caused by changing c to $\tilde{c}$ is bounded by

$$|\Delta R_+| \doteq \frac{|\Delta c|}{2|aR_+ - b|} \lesssim \frac{\epsilon|c|}{|aR_+ - b|} = \frac{2\epsilon|R_+R_-|}{|R_+ - R_-|}$$

if ϵ^2 terms are ignored. These bounds are almost rigorous; by applying results from Smith [1] or Börsch-Supan [2] we may verify the first few formulae in fig. 7, which provide rigorous *a posteriori* bounds for $|\Delta R|$. These bounds do not assume anything about the source of the approximations; why don't we just use these bounds and skip the foregoing rounding error analysis? There are three reasons why.

First, a rounding error analysis, even if not entirely rigorous, indicates how likely are the computed values to repay the cost of their computation. Without that analysis we must wait until after the computation to discover whether it was worthwhile; could we perhaps get a better answer sooner by repeatedly invoking a random number generator until its output satisfies acceptable *a posteriori* bounds?

An *a posteriori* bound for roots:

Let r_+ and r_- be given approximations to the roots of

$$P(z) \equiv z^2 - 2bz/a + c/a = 0 .$$

If $r_+ \neq r_-$ then each of the two regions

$$|z-r_\pm| \leqslant 2P(r_\pm)/|r_+-r_-|$$

in the z-plane contains one of the roots of $P(z) = 0$ unless those regions overlap, in which case their union contains both roots. If $r_+ = r_- = r$ the region

$$|z-r| \leqslant \tfrac{1}{2}|P'(r)| + \sqrt{\tfrac{1}{4}|P'(r)|^2 + |P(r)|}$$

contains both roots. (Here the prime means derivative.)

An *a priori* bound for roots:

If the roots of $ax^2 - 2bx + c = 0$ are $R_\pm$, and the roots of $ax^2 - 2bx + c(1+\gamma) = 0$ are $\tilde{r}_\pm$, then the relative differences $\delta_\pm \equiv 1 - r_\pm/R_\pm$ are bounded by

$$|\delta_\pm| \leqslant \sqrt{|\gamma|}\left\{\sqrt{1+|\gamma|}+\sqrt{|\gamma|}\right\}.$$

Fig. 7. Bounds for perturbed roots.

Secondly, *a posteriori* bounds frequently cost at least about as much as the computation they are intended to validate, and more if no advantage is taken of what might reasonably be inferred about the role of roundoff in that computation. Furthermore, the computation of bounds is another computation susceptible to rounding errors. For example, when the expression $(A*Z-2.*B)*Z + C$ is computed using the coefficients A, B, C and 4-significant decimal rounded arithmetic of fig. 3, it vanishes for $Z = .9860$, for $Z = 1.011$, and for several other 4-significant decimal values between them, despite the fact that the intended quadratic should vanish only twice (i.e., at $Z = .9974_{7422}$ and $Z = 1.0$). Evidently, the *a posteriori* bounds of fig. 7 cannot be applied to the computed values of $P(r_\pm)$ unless either those values are computed more precisely (is double-precision arithmetic obviously good enough?), or else those values are reconciled with roundoff. The quadratic expression above is approximated in storage by a computed value

$$(\{(az(1+\mu_1)-2b(1+\mu_2))/(1+\alpha_1)\}z(1+\mu_3)+c)/(1+\alpha_2)$$

in which the Greek letters represent, as before, rounding errors bounded by ϵ; by means discussed in Adams [3] we may compute that roundoff contributes roughly as much uncertainty as if the value 47.39 of C were uncertain by about two ulps. Consequently, after four-digit calculations provide estimates $(0.02\pm0.02)/47.51$ for both values of $P(r_\pm)$, the best inference from fig. 7 places both desired roots' real parts somewhere between 0.938 and 1.06, and imaginary parts between ±0.04; these bounds are not worth the effort to compute them*.

Cheaper bounds can be achieved by doing more analysis first and then less computation. For example, from

$$|P(\tilde{r}_\pm)| = |c-\tilde{c}|/|a| \lesssim 2\epsilon|c/a| \quad \text{and} \quad |\tilde{r}_\pm - r_\pm| \lesssim 5\epsilon|r_\pm|$$

we may conclude via fig. 7 that the desired roots lie in the union of the two regions

$$|z-r_\pm| \lesssim 5\epsilon|r_\pm| + 4\epsilon|c/a|/\{|r_+-r_-|-5\epsilon|r_+|-5\epsilon|r_-|\} ,$$

which place real parts between 0.921 and 1.08, imaginary between $\pm .06$. These cheap bounds are poor too, but better bounds are nearby.

A third reason for not skipping the analysis of roundoff is that it provides better bounds. Aware that a perturbed quadratic $ax^2 - 2bx + c$ exists, we may invoke the *a priori* bound in fig. 7; its proof follows lines laid down by Ostrowski ([4], Appendix B). After inferring $|R_\pm - \tilde{r}_\pm|/|r_\pm| \lesssim \sqrt{2.1\epsilon}$ and recalling $|r_\pm - \tilde{r}_\pm|/|\tilde{r}_\pm| \lesssim 5\epsilon$ we deduce that each desired root $R_\pm$ lies in a circle

$$|R_\pm - r_\pm|/|r_\pm| \lesssim \sqrt{2.1\epsilon} + 5\epsilon \doteq .035$$

(ignoring ϵ^2 terms). Despite the fact that these circles overlap, each contains one root. Consequently, if both roots are real

$$0.943 \leqslant R_- \leqslant 1.02 \quad \text{and} \quad 0.985 \leqslant R_+ \leqslant 1.06$$

whereas if they are a complex conjugate pair $R_\pm = R_r \pm iR_i$

$$0.985 \leqslant R_r \leqslant 1.02 \quad \text{and} \quad -0.03 \leqslant R_i \leqslant 0.03 .$$

Though better than before, these bounds are still three times wider than they could be.

The foregoing few paragraphs are not intended to disparage *a posteriori* error bounds; these bounds are invaluable for validating results of long calculations, and for sensitivity analyses. For example, if our coefficients A, B, C are uncertain by, say, 5 ulps each then $P(r_\pm)$ must be uncertain by roughly $\pm0.15/47.51$ and the desired roots must be uncertain to an extent not grossly overestimated via fig. 7, namely

$$|R_\pm - r_\pm| \lesssim 2(0.15/47.51)/|r_+ - r_-| \doteq 0.15 .$$

But when A, B, C are known precisely the *a posteriori* techniques may be hampered by a restriction to arithmetic no more precise than was used to compute the approximations under test; their bounds may be no better than if A, B, C were uncertain by about an ulp each.

In our example the limitations of 4-digit arithmetic can be circumvented by an old trick; observe that the substitution $x = 1 + y$ changes $47.51x^2 - 2 \times 47.45x + 47.39$ into a new quadratic $47.51y^2 + 2 \times 0.06y + 0$ whose coefficients happen to be computable precisely with 4-digit arithmetic. We shall return to this trick later.

6. HASTY JUDGEMENTS

"The Purpose of Computing is Insight, not Numbers."(1962)
"The purpose of computing numbers is not yet in sight,"(1970)

R.W.Hamming

At this point the tired reader may be tempted to draw from the foregoing mass of arithmetic some wrong conclusions:

1. Error analysts are nit-pickers who delight in finding last-figure errors in other error analysts' calculations, and don't do much else. This may be true, but it is not the right conclusion.

2. Since error analysts cannot solve a problem as given, but must first imagine it to have been altered by an ulp or two here and there, they cannot legitimately protest when the arithmetic unit of an electronic computer produces results no more wrong than if every operand were first perturbed by an ulp. That this is quite wrong will be apparent later.

* A more delicate analysis shows that, for the values A, B, C under discussion here, $\mu_2 = \alpha_1 = \alpha_2 = 0$, whence improved estimates for $P(r_\pm)$ are $(0.02\pm0.01)/47.51$, and 0.948, 1.05 and ±0.02 for the roots' bounds. But only a well-implemented Interval Arithmetic program is capable of such delicacy.

3. A principal source of error in numerical computation is cancellation, which should therefore be avoided or circumvented whenever possible. This is wrong too because cancellation cannot create error despite contrary appearances in figs. 1 and 3; moreover, artful cancellation can help diminish error, as we shall see.

The correct conclusion is this:

Error analyses, especially those concerned with roundoff, are so tedious, so much nastier than the calculations they are intended to validate, and so frequently unrewarding, that they should not be inflicted inconsiderately by one man upon another.

Why, then, inflict such an analysis upon the reader?

My motive now is the same as it was when I reported [5] on modifications to the IBSYS operating system on the University of Toronto's IBM 7094-II and their impact upon a library of numerical subprograms:

> "... users of these subprograms need not supplement their own competency in mathematics, science, engineering or the humanities by a hyperfine proficiency at both numerical analysis and the debugging of systems programs..."
>
> "For as long as electronic computers have been in use (since 1949 at the University of Toronto), there has existed a steadfast policy to widen the range of intellectual disciplines that might benefit from the machine. That policy is partly responsible for a decline in the numerical sophistication of users, a decline which has yet to be compensated by an increased sophistication in the programs they can use. Despite intensive attempts to educate them in the arts of computation, too many new users attribute to the numerical library subprograms the infallibility of a mathematical proof. They shall be disillusioned. To what extent can their disillusionment be written off as part of their education? To what extent can their dissatsfaction be traced to shoddy computing systems? There is room for improvement in both the quality of education and the quality of computer performance. But you cannot teach an old dog new tricks, and you cannot teach a new dog very much. Therefore the bulk of the improvement must and can come in the performance of computer systems."

From a numerical analyst's point of view computer systems have improved mainly in speed and storage capacity since those words were written, but have deteriorated in several other respects. Of course, there are exceptions. For example, the elementary function subroutine library* supplied for Fortran on IBM System/360 machines by Hirondo Kuki of the University of Chicago is a triumph of persistent diligence over the nastiness of hexadecimal arithmetic, but according to Cody [6] the high quality of that library is atypical of current commercial practice. Furthermore these subprograms, like other packages of scientifically oriented subprograms distributed variously by computer systems' manufacturers, user organizations like SHARE, software firms, universities and other major research centers, tend to be closely tuned to some specific machine or operating system and go out of tune when moved. The same is true of some of the ostensibly machine-independent programs published in various journals of computing and numerical analysis. The fault rarely lies in those programs as published; more often it lies in a computer system described as "compatible with XXX (except for YYY)". Wherever the fault may lie, the result is the same; the computer user is obliged to learn more about the details of the programs and of his computer system than he had intended.

What would happen to our society if everybody who wished to use a telephone, a television set, a car, a detergent, a plastic toy or a computer were obliged first to learn at least a little about how it was made and how it works internally, and then to test it himself for hazards and other surprises?

An environment in which a computer program can operate reliably on any of several computer systems can be achieved partly by a measure of standardization, but mostly requires that attention to detail which, by eliminating anomalies and arbitrary restrictions, promotes economy of thought. The assertion that a program is machine-independent and reliable is worthless if it is not susceptible to both analytical and experimental verification. Here is where error analysis can make its contribution, not so much by providing error bounds for specific numerical procedures as by providing a rationale which, when combined with an harmonious computing environment, assures that such bounds will be found without exorbitant intellectual effort.

Computer systems, hardware and software, are not coming into harmony with the rationale of error analysis. I shall support this contention with examples. The examples are contrived; they are artificial because the complications of real computations tend to dis-

* Some of these programs are described in IBM System/360 *Fortran IV Library Subprograms,* Form C28-6596, and others in Kuki and Ascoly [7].

tract attention from the roots of disharmony. They are designed to show why error analysis on today's computer systems is turning into necromancy. If they help hardware and software designers learn a little more about error analysis, and if error analysts learn a little more about hardware and software, and if we collaborate, we can re-establish error analysis as a humdrum scientific activity from which most computer users may safely be spared.

7. BACK TO THE QUADRATIC

We saw that fig. 2's program approximates, to within a few ulps, the roots of a quadratic equation whose coefficients match the given coefficients to within a few ulps. From this we inferred, without further reference to that program, that the computed roots match the "true" roots to at least about half as many significant figures as were carried during the computation. Since a program which loses half the figures carried seems less than exemplary, we are led to three questions:

1. Is the error analysis realistic?
2. If so, can the program be improved?
3. If so, is the improvement worth its cost?

We shall see that the answers are respectively:

1. Yes.
2. Yes, on most computers.
3. Yes, on some computers, in some dialects of Fortran.

That the error analysis is realistic follows from the sharpness of the assertions in fig. 7; Smith [1] has shown the *a posteriori* bounds there to be pessimistic by factors not much larger than 2, and the *a priori* bound's inequality becomes equality when $\gamma > 0$ and $\tilde{r}_+ = \tilde{r}_-$. Hence it follows that, however many figures the program may carry, examples like fig. 3 must exist for which half the figures are lost. The loss can be traced to those rounding errors μ_1 and μ_2 in figs. 4 and 5 which are interpretable as perturbing the coefficient c. Were those perturbations μ_i with $|\mu_i| \leqslant \epsilon$ replaced by smaller $|\mu_i| \leqslant \epsilon^2$, whence the new perturbed coefficient c would satisfy $|c-\tilde{c}| \leqslant 2\epsilon^2|c|$, the *a priori* bound in fig. 7 would lead to new bounds like

$$|R_\pm - r_\pm|/|r_\pm| \leqslant \sqrt{2.1\epsilon} + 3.6\epsilon$$

instead of the previous $\sqrt{2.1\epsilon} + 5\epsilon$. In other words, roots accurate to nearly single precision could be obtained by evaluating the products $B**2$ and $A*C$ and subtracting them in double-precision before rounding the result to a single-precision D in fig. 2.

Despite the fact that the hardware of many computers provides easy access to the precise double-length product of two single-precision numbers, today's programming languages tend to obstruct that access, and future hardware designs could respond to its consequent disuse by eliminating it. For example, in the older dialects of Fortran IV on the IBM 7094 (IBSYS versions up to 12) we could get what we wanted by replacing

```
D = B**2 - A*C
```

in fig. 2 by

```
DOUBLE PRECISION DD
DD = B*B
D = DD - A*C
```

The old compilers recognized a double-precision context in which truncation of $B*B$ and $A*C$ to single-precision did not occur. Today's compilers obstinately truncate, thereby producing a result no better than if *DD* were merely a single-precision variable. To achieve what we want now we must write

```
D = DBLE(B)**2 - DBLE(A)*DBLE(C) ,
```

which appends zeros to the right of A, B, and C's values and goes through the wasted motion of two full double-precision multiplications.

While at the University of Toronto, I circumvented this foolishness by adding a built-in function *DSIC* to our Fortran compiler, thereby permitting simply

```
D = DSIC(B*B) - DSIC(A*C)
```

to yield the desired result. *DSIC* accepted simple sums and products of single-precision variables and produced their doubly-precise evaluation. This function found wide application, especially for doubly-precise accumulation of scalar products of single-precision vectors, and rendered many matrix handling programs more nearly transparent by freeing them both from machine-language subroutines intended to accomplish the same effect and from subtle errors induced by arbitrary and easily forgotten implicit parsing rules. *DSIC* was very fast on the 7094's Fortran IV version 12 since no superfluous instructions were generated; some of this speed was lost during the transition to version 13.

```
      PROGRAM SILLY (INPUT,OUTPUT,TTYOUT,TAPE1=TTYOUT)
      X = 1.0 + 3/2
      Y = 1.0 + (3/2)
      WRITE (1,1)  X, Y
1     FORMAT(/9X,*1.0 + 3/2 = *,F5.2,6X,*1.0 + (3/2) = *,F5.2,/)
      STOP
      END
:

BEGIN EXECUTION     SILLY
          1.0 + 3/2 =  2.50       1.0 + (3/2) =  2.00
STOP       SILLY
>
```

Fig. 8. Never underestimate the power of parentheses.

The issues at stake here go beyond convenience and efficiency; they bear upon our ability to say what we mean or mean what we say when we use programming languages. For example*, in PL/I we find

$25 + 1/3 = 5.333...$ with *FIXEDOVERFLOW*,
but
$25 + 01/3 = 25.333...$.

One of the Fortran dialects used on CDC 6000-class machines allows mixed-mode integer and real arithmetic to give the results shown in fig. 8, which was taken off a terminal connected to Berkeley's 6400. Some compilers cause different values to be assigned to

$Y = X + 3.14159$ and $Z = X + 3.1415900000$,

whereupon arithmetic comes to depend not upon the values of numbers but upon accidents of notation, as if we could divine something more than its value from a number by looking at the way it is written.

Despite the ascendancy of computers, mankind will continue to hold that

$3.14159 = 3.1415900000 = 3.1415900000...$

$= 314159/100000$,

and none of these digit strings is correctly a substitute for the transcendental $\pi = 3.14159\ 26535...$ or for the interval $[3.14158\ 5, 3.14159\ 5]$ or for the integer 3, nor can the unique rational number they represent be represented by a single binary floating point number in a computer. Of course approximation is necessary, but when one number in hand must be approximated by another the approximation should ideally depend upon the *value* of the first number and upon the context in which the second will be used, not upon how many digits are alleged to be "significant". These notions have been explained lucidly by DeLury [40] and are realized in Algol on at least some computers (e.g., Burroughs B5500).

In a properly designed computing environment, both digit strings 3.14159 and 3.1415900000 should be converted to the same binary approximation in otherwise indistinguishable contexts; whether they are approximated to single- or to double-precision should depend *only* upon that context. Similarly, whether the computed *value* of $A*C$ will be retained in double-precision or rounded to single-precision should depend upon the context in which it appears and *not* upon the ostensibly single-precision formats of A and C, whose values may, like 3.0, be in no way imprecise. We should have the option to round $A*C$'s value to single-precision by writing, say,

RND(A*C)

as I used to do at Toronto. Then the language designer can choose any convenient and simple conventions whereby implicit *RND*s or *CHOP*s or *DSIC*s may be understood to be compiled into any expression in appropriate places; e.g., when we write simply

D = B**2 – A*C

we may read

D = CHOP(CHOP(B*B)–CHOP(A*C)) .

* This example is drawn from p. 231 of *IBM System/360 Operating System PL/I(F) Language Reference Manual*, File #S360-29, GC28-8201-3.

And if we dislike what we read we may write instead

D = DSIC(B*B) – DSIC(A*C)

and read

D = CHOP(DSIC(B*B)–DSIC(A*C)) ,

to which now corresponds a computed value

$d = (1+\sigma)(b^2-ac)$ with $|\sigma| \leqslant \epsilon$.

This last equation is not quite accurate. It would be true (if *DSIC* were implemented there) on IBM System/360 machines now that they retain a guard digit for double length arithmetic. But the 7094, like most other computers, does not retain such a guard digit, and consequently may discard prematurely the last few digits of the smaller of two double-precision numbers being subtracted, though the difference will not then be in error by more than if instead the larger number were first altered by one ulp of double precision. This corresponds to computing (1.0–0.9999 9999) using "eight significant figure arithmetic" in one of the following ways:

(like IBM 7094, double precision)	*(like CDC 6400, single precision)*
1.000 0000	1.000 0000
–0.999 9999 9	–0.999 9999 9
$0.000\ 0001 \rightarrow 10^{-7}$	$0.000\ 0000\ 1 \rightarrow 0$

Doing arithmetic this way is sometimes excused by the argument, which we shall demolish later, that nobody can say exactly what the last digit of a high-precision number ought to be, so nobody should care if it is altered a little.

It appears that the value computed for D above will satisfy

$d = (1+\sigma)\{(1+\mu_1)b^2-(1+\mu_2)ac\}$

with

$2|\mu_1| \leqslant \epsilon^2$, $2|\mu_2| \leqslant \epsilon^2$, $\mu_1\mu_2 = 0$ and $|\sigma| \leqslant \epsilon$.

(The factors 2 are appropriate for the 7094, a binary machine, with $\epsilon = 2^{-26}$ for chopped arithmetic.) The final result does not seem to deteriorate much; we get

$|R_\pm - r_\pm|/|r_\pm| \leqslant \sqrt{1.1}\epsilon + 4.1\epsilon < 5.2\epsilon$

for the relative error in the computed roots. However, when the computed roots are complex with relatively tiny imaginary parts we may wonder whether those tiny numbers are accurate to nearly full single precision. They are; this does not follow from the inequalities for μ_1 and μ_2 given above but can be proved laboriously to be true for every major North American computer with double-precision hardware; the reader is urged to try to prove this claim for his own computer.

We are now almost in a position to which every conscientious error analyst aspires from time to time. We have a program which will solve a familiar problem accurately, at a cost (on decent computer systems) which is scarcely more than minimal, without having to inflict upon our program's users any more of our error analysis than the following simple statement:

Given the single precision coefficients A, B, C of the quadratic equation $Ax^2 - 2Bx + C = 0$, the program computes the roots correct in every respect to within a few (10 on an IBM 7094) units in the last place quoted, *except for over/underflow.*

8. OVER/UNDERFLOW

> Oh, the little more, and how much it is!
> And the little less, and what worlds away!
>
> *By the Fire-side*
> Robert Browning

An earlier report [5] describes modifications done to the IBSYS operating system, on the IBM 7094-II at the University of Toronto, which were designed to shield ordinary computer users from the nuisance of those over/underflows which could reasonably be suppressed, circumvented or ignored automatically by a well designed computer system. After the modifications were introduced, most over/underflows became invisible to users, provably exerting no adverse effect upon their computations, and the persistent over/underflows were rendered relatively easy for each user to locate and cure as he pleased. I have the impression that over/underflow became far less of a nuisance on Toronto's IBM 7094, despite its normal*

* The modifications included provision for certain kinds of Fortran calculations to be carried out efficiently and conveniently with magnitudes as extreme as 10.0**(±10**12), but these were rarely used.

number range of 10^{-38} to 10^{+38}, than it is now on Berkeley's CDC 6400 with a far wider range of 10^{-294} to 10^{+322}. The reader may form his own impression by comparing what he must do on his computer with what we used to do on the 7094 to cope with over/underflow when solving quadratic equations.

Our object is to replace the phrase "*except for over/underflow*" above by this statement:

Overflow is reported if and only if a result must overflow, and similarly for underflow, and over/underflow in one result does not degrade the accuracy of the other.

A program matching these specifications is surprisingly useful. Quadratics with exorbitantly large or small coefficients arise, for example, when solving large dimensional determinantal equations by certain iterative methods, and the fact that those coefficients may easily be re-scaled to reasonable magnitudes is no excuse for not doing so in the program which solves the quadratic. Failure to re-scale the coefficients can lead to over/underflow during the computation of D in fig. 2, and hence give no solution or else a wrong one. Furthermore, occasions arise when one seeks a distinguished root of a quadratic whose coefficients depend upon a parameter in such a way that the unwanted root tends to zero or infinity; this is why we do not want over/underflow in one root to contaminate the other.

Here is one of the algorithms that work. First dispose of the possibilities $a = 0$ or $c = 0$. Then choose h to be a power of the radix (2 on the 7094) such that neither a/h nor c/h over/underflows and yet $|(a/h)(c/h)|$ lies relatively close to 1, say between $\frac{1}{4}$ and 4. (The best choice for h is not worth discussing here.) We used to compute h in various ways, sometimes by tricky machine-dependent integer-arithmetic manipulations of a and c, sometimes by logical bit-manipulations, but always by means available through our version of the Fortran compiler. Next compute $\bar{a} \equiv a/h$, $\bar{c} \equiv c/h$ and $\bar{b} \equiv b/h$. If $\bar{b}$ or $\bar{d} \doteq \bar{b}^2 - \bar{a}\bar{c}$ overflows, suppress that overflow indication and produce $r_+ \doteq b/(\frac{1}{2}a)$ and $r_- \doteq (\frac{1}{2}c)/b$ as roots of $ax^2 - 2bx + c = 0$. If $\bar{b}$ or $\bar{b}^2$ underflows, suppress that underflow indication and replace $\bar{b}$ by zero and continue. Otherwise, compute the roots as usual using $\bar{a}$, $\bar{b}$, $\bar{c}$ in place of a, b, c. Remember that $\bar{d}$ must be computed with a double precision subtraction. Each root will be computed as a final quotient in which no over/underflow can occur unless it is very nearly unavoidable and must be reported.

The only loose end in the foregoing algorithm is how to choose h, which we shall leave loose with the observation that h can generally be constructed easily and quickly in Fortran and in machine language, but not so quickly in Algol. We must also suppress irrelevant over/underflow signals, and enable the relevant ones; here is where the advantages of the 7094 system became apparent, because they involved few explicit tests and almost no loss of time. One complete program to solve a quadratic properly took less than 20% longer to execute than did a naive program based upon fig. 2.

The algorithm is expensive to implement on a CDC 6400 for several reasons. First, the machine gets confused when asked whether a number is zero or not (see fig. 9) because it sometimes tests only the first 12 instead of the first 13 bits of a floating number (see CDC's 6400/6500/6600 Computer Systems Reference Manual, Pub. no. 60100000, rev. A (1969), pp. 3–18). Secondly, the machine sets underflowed numbers to zero without any warning indication; this causes problems like that in fig. 10 where the value of Y differs from 1.0 by rather more than could be attributed to 11 rounding errors. Thirdly, many tests are required, one after each arithmetic operation susceptible to overflow, in order to avoid being kicked off the machine for attempting to use arith-

```
      PROGRAM NAUGHT (INPUT,OUTPUT,TTYOUT,TAPE1=TTYOUT)
      Z = 0.5**976
      ZZ = Z+Z
      IF( Z .NE. 0.  .AND.  Z*100. .EQ. 0.  .AND.
     *      Z/0.01 .EQ. 0. )   WRITE(1,1)  Z, ZZ
    1 FORMAT(44H  Z .NE. 0.  BUT  Z*100. = Z/0.01 = 0.  AND  , /
     *  * PRINTING YIELDS  Z = *, 1PE12.4, * ,  Z+Z = *, 1PE12.4 )
      STOP
      END
$

 BEGIN EXECUTION     NAUGHT
   Z .NE. 0.  BUT  Z*100. = Z/0.01 = 0.  AND
   PRINTING YIELDS  Z =   0.          ,  Z+Z =   3.1315-294
 STOP       NAUGHT
```

Fig. 9. Is Z zero or naught?

```
      PROGRAM  WHY (INPUT,OUTPUT,TTYOUT,TAPE1=TTYOUT)
       Z = 2.0**(2**10 - 48)
       C = 1.0/Z
       A = C+C
       B = A*10.0**9
       D = A+B
       X = (B+D)/A
      Y = ( (A*X+B)/(C*X+D) )/( (A+B/X)/(C+D/X) )
      IF( A.GT.0. .AND. B.GT.0. .AND. C.GT.0. .AND.
     *    D.GT.0. .AND. X.GT.0. .AND. Y .GT. 2.999
     *   )   WRITE (1,1)   Y
     1  FORMAT(  5X, *WHY DOES  Y = *, F15.11, *  ?* )
       STOP
       END
:


 BEGIN EXECUTION      WHY
     WHY DOES  Y =   2.99999999875  ?
 STOP       WHY
```

Fig. 10. Why is Y so far from 1.0?

metically a previously overflowed result. The machine can also operate in a mode which allows continued operation upon "infinities" and "indefinites", but this liberal mode is rarely used and cannot be invoked nor repealed from within a Fortran program. The reason why the liberal mode is rarely used may be that any rules for manipulating the symbols ∞ (infinity) and θ (indefinite) must be potentially misleading; the following example compares what should be expected with what the 6400 actually computes.

Program	*Expected values*	*Observed values*
X = 2.0**1069	2^{1069}	2^{1069}
Y = 4.0*X	2^{1071}	∞
Z = Y−2.0*(X+X)	0	θ
T = (((Y−X)−X)−X)−X	0 or θ	∞ !
U = 1.0/T	∞ or θ	0 !
V = X/Y	$\frac{1}{4}$ or θ	0 !

Finally, CDC's Fortran compilers have nothing equivalent to *DSIC*, and one must use DBLE inefficiently instead.

If numbers like 10^{300} were sinful and numbers like 10^{-300} obviously negligible, the design of the 6400 would make sense. But why draw the lines there instead of at 10^{150} and 10^{-150}? If over/underflow is so obvious a mistake, why does it happen to experienced professionals like Fettis and Caslin [8]?

Integer overflow reveals another notorious defect in most compiler designs, as Korfhage [9] could testify. On the CDC 6400 the defect is enshrined in hardware which gives no indication of integer overflow. In fig. 11, obtained from our 6400, every arithmetic expression is computed correctly, but J is incorrectly compared with K because J−K overflows. Fig. 12 has two programs which differ only in that*

DO 2 N = 1, L, 1

has been replaced by

INCREM = 1

DO 2 N = 1, L, INCREM .

The mysterious diagnostic tells the programmer that he has abused the computer, but does not tell how. It turns out that a division by zero occurred in the first program's statement 2. All can be explained by the observation that integer arithmetic in CDC's Fortran is carried out sometimes modulo $2^{17} - 1$,

* The terminal symbols ",1" could be deleted without altering the results.

```
      PROGRAM GOOF (INPUT,OUTPUT,TTYOUT,TAPE1=TTYOUT)
      I = 2**40
      DO 11   L = 1, 18  .
 11       I = 1+I
      J = I + 3
      K = -I
      IF( J .GT. 0  .AND.  K .LT. 0  .AND.  J+K .EQ. 3
     *      .AND.  J .LT. K  )   WRITE (1,1)
     1  FORMAT(*  WHY IS  0 < (I+3) < -I < 0  ?*)
      STOP
      END
:


 BEGIN EXECUTION      GOOF
   WHY IS  0 < (I+3) < -I < 0  ?
 STOP       GOOF
```

Fig. 11. Integers out of order.

```
      PROGRAM MUDDLE (INPUT, OUTPUT, TTYOUT, TAPE1=TTYOUT)
C     TO COMPUTE THE INFINITE SUM OF  N/(1 + N**3) FOR N = 1, 2, 3,...
C     CORRECT TO 10 FIGURES, JUST ADD THE FIRST 300000 TERMS AND AN
C     EULER-MACLAURIN CORRECTION .
      EPS = 0.1**10
      L = 3.0/SQRT(EPS)
      WRITE (1,1)  L
1     FORMAT(3X,11H THE SUM OF , I7, 24H TERMS  N/(1 + N**3)  IS )
      SUM = 0.
      DO 2   N = 1, L, 1
        EN = N
2       SUM = SUM + EN/(1.0 + EN**3)
      WRITE (1,3)  SUM
3     FORMAT( 13X, F16.12, /)
      SUM = SUM + 1.0/EN
      WRITE (1,4)  SUM
4     FORMAT(3X,21H THE INFINITE SUM IS  /  3X,F16.12)
      STOP
      END
:

 BEGIN EXECUTION     MUDDLE
     THE SUM OF 300000 TERMS  N/(1 + N**3)  IS
 USER CPU ARITH-ERROR
 I:  DETECTED BY MTR  , FL = 007455
 >


      PROGRAM FIDDLE (INPUT, OUTPUT, TTYOUT, TAPE1=TTYOUT)
C     TO COMPUTE THE INFINITE SUM OF  N/(1 + N**3) FOR N = 1, 2, 3,...
C     CORRECT TO 10 FIGURES, JUST ADD THE FIRST 300000 TERMS AND AN
C     EULER-MACLAURIN CORRECTION .
      EPS = 0.1**10
      L = 3.0/SQRT(EPS)
      WRITE (1,1)  L
1     FORMAT(3X,11H THE SUM OF , I7, 24H TERMS  N/(1 + N**3)  IS )
      SUM = 0.
      INCREM = 1
      DO 2   N = 1, L, INCREM
        EN = N
2       SUM = SUM + EN/(1.0 + EN**3)
      WRITE (1,3)  SUM
3     FORMAT( 13X, F16.12, /)
      SUM = SUM + 1.0/EN
      WRITE (1,4)  SUM
4     FORMAT(3X,21H THE INFINITE SUM IS  /  3X,F16.12)
      STOP
      END
:

 BEGIN EXECUTION     FIDDLE
     THE SUM OF 300000 TERMS  N/(1 + N**3)  IS
                1.111640603830

     THE INFINITE SUM IS
      1.111643937163
 STOP       FIDDLE
 >
```

Fig. 12. What did the first program DO wrong?

sometimes modulo 2^{48}, and sometimes modulo $2^{59} - 1$, depending upon the whims of the compiler.

Incidentally, although the series has been summed using 48 significant bit (about 14 decimal) arithmetic, the two 13-decimal numbers printed out have been contaminated by roundoff in their last 4 digits; the correct values are 1.1116 4060 *4896* and 1.1116 4393 *8230* respectively.

I hope the reader will not think that I think computers are conspiring against me alone; that would be a paranoid delusion.

9. A HORROR STORY

> "... lo mal fabbro biasima lo ferro..."
> (... the bad blacksmith blames the iron...)
>
> *Convivio* I *xi*
> Dante Alighieri

Mr. Z. was despondent when I first saw him. A graduate student of aeronautical engineering, he was trying to augment boundary layer flow past wings in

a way which might enhance their lift at low speeds. If his idea worked, his reward would be a Ph. D. thesis and a job with a local firm designing STOL aircraft. He was testing his idea on our university's computer, then an IBM 7090, by solving numerically a complicated system of differential equations, finally producing a graph from which he could read Success or Failure. He had just read Failure.

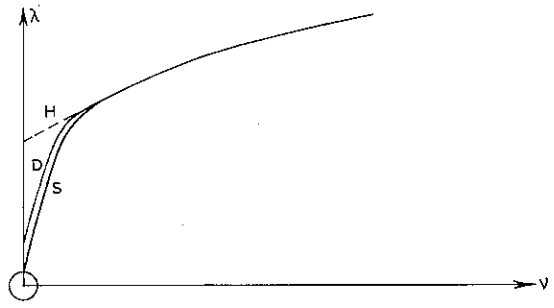


Fig. 13. Mr. Z.'s graphs: H is the graph he Hoped to get, S is the graph produced by Single-precision computation, D is the graph produced by Double-precision computation.

Fig. 13 is a simplified picture of his program's output. The close agreement between single- and double-precision results, and their disagreement with his expectations, seemed to prove conclusively that he should look for a new thesis topic.

At that time I was testing an intended replacement for IBM's single precision logarithm subroutine. Of course, I had proved mathematically that my new subroutine was preferable to IBM's in every way, but a vestige of self-doubt induced me to re-run several users' programs with my logarithm substituted for IBM's. Mr. Z.'s program was one of those re-run, and one of very few whose results were altered appreciably by the substitution. His graph S moved to position H. I was alarmed because I had expected my improved subroutine to produce single-precision results closer to double-precision, not further away; and Mr. Z. was surprised because he had no explicit reference to logarithms in his Fortran program. We soon discovered where a logarithm lurked in his program; it was in a sub-routine which I have simplified and listed in fig. 14.

Here is an outline of Mr. Z.'s error analysis of his program to compute $F(X,G) = X^{G(X)/(X-1)}$ for $X > 0$. He established first that $G(X)$ was well-behaved; $0 < G(X) \leqslant \sqrt{X}$ and $|\mathrm{d}\log G(X)/\mathrm{d}\log X| \leqslant 2$. Next he checked that the computed value $g(x)$ differed from $G(x)$ by at most an ulp or two: $g(x) = (1 \mp \gamma)G(x)$ for some tiny relative error γ. Then he verified that defining

```
      FUNCTION F(X,G)
C        Given a function G(X) well-behaved for all X > 0,
C        this FUNCTION subroutine computes
C        F(X,G) = X^(G(X)/(X-1)) correctly to within a few
C        ulps.
    1 IF (X .LE. 0.0) Complain "F(X,G) undefined for X ≤ 0"
    2 IF (X..EQ. 1.0) F = EXP(G(X))
    3 IF (X .NE. 1.0) F = X**(G(X)/(X-1.0))
      RETURN
      END
```

Fig. 14. Mr. Z.'s subroutine.

$$F(1,G) \equiv \lim_{x\to 1} F(x,G) = \exp(G(1))$$

made $F(X,G)$ continuous for all $X > 0$, and bounded $(1 < F \leqslant \exp(1) < 2.72)$ and, most important, $|\mathrm{d}\log F(X,G(X))/\mathrm{d}\log X| \leqslant 3$. Now he knew that $F(X,G(X))$ was a "well-conditioned" function of X in the sense that relatively small variations in the argument X could not cause much larger relative variations in F. Specifically, whenever the value x stored in the cell called X was a good approximation to the intended value X, then the value $F(x,G(x))$ would closely approximate $F(X,G(X))$. All that remained was to show that roundoff during the computation of what was intended to be $F(X,G(X))$ would produce a computed value f relatively close to $F(x,G(x))$.

He observed that writing (X−1.0) caused $(1-\sigma)(x-1)$ to be computed, with σ representing a rounding error smaller than 1 ulp of $(x-1)$. Similarly, the expression G(X)/(X−1.0) would introduce another rounding error δ into the computed quotient, producing

$$\begin{aligned} y &\equiv (1-\delta)g(x)/\{(1-\sigma)(x-1)\} \\ &= (1-\delta)(1+\gamma)G(x)/\{(1-\sigma)(x-1)\} \\ &= (1+\eta)G(x)/(x-1)\,, \quad \text{say,} \end{aligned}$$

where η represents an accumulated error, due to roundoff, of at most a few ulps. Now he made his first mistake; he assumed that writing X**Y in Fortran would produce a computed value $(1+\rho)x^y$ in which ρ represents another error, due to roundoff, of at most a few ulps. Had that assumption been true, his conclusion, that the computed value

$$f = (1+\rho)x^y = (1+\rho)F(x,G(x))^{1+\eta}$$

matched $F(x,G(x))$ and hence $F(X,G(X))$ to within a few ulps, would have been correct. His second mistake was to test his program on only 31 values of X distributed uniformly between $X = 0.5$ and $X = 2.0$ and on about as many values of X outside that interval, these tests could not reveal his first mistake.

Why was his assumption about X**Y wrong? It would have been correct for a log-log slide rule, but at that time our 7090 obtained X**Y by computing EXP(Y*ALOG(X)), and the logarithm program then (as on many other computers now) produced not $\log x$ but $(1+\lambda)\log\{(1+\xi)x\}$ with λ and ξ each representing errors of about two ulps. The error ξ was introduced through the familiar formula

$$\log x = \log((1+z)/(1-z)) - \tfrac{1}{2}\log 2 \quad \text{with}$$

$$z \equiv (2x-\sqrt{2})/(2x+\sqrt{2}),$$

because the value stored for $\sqrt{2}$ was rounded and also z was rounded. The end result was to compute $f \doteq F^{1+\xi/\log x}$ instead of F, and this result was very wrong whenever x differed from 1 by only a few ulps.

My new logarithm subroutine* took care to keep $\xi = 0$, caused X**Y to be approximated by $(1+\rho)x^y$ as expected, and allowed Mr. Z.'s program to give the results he desired in single-precision. But why were his double-precision results different? At first we thought the double-precision DLOG program contained a flaw too, but it turned out to be unexceptionable. Then IBM issued a revision to the double-precision package on the 7090 which made graph D go away; new graphs computed in both single- and double-precision confirmed Mr. Z.'s hopes and he was happy. For a while.

A few months later the 7090 was replaced by a 7094 with built-in double-precision hardware, and graph D came back. We soon discovered that the double-precision subtraction hardware on the 7094 lacked a guard bit which the 7090's latest software had preserved. Consequently, when $x-1$ was computed on the 7094 for x slightly less than 1, the hardware first discarded x's last (54th) bit and then did the subtraction. The resulting value f approximated not F, as desired, but $F^{1/2}$ or $F^{2/3}$ or $F^{3/4}$ or ... depending upon x's last few bits. Mr. Z. cured this problem by substituting the expression ((X−0.5)−0.5) for (X−1.0) in his program, which is now machine-independent and runs correctly on any computer system with respectable exponential and logarithm subroutines.

Was Mr. Z. clever or just lucky? How often are engineers baffled by subtly wrong computations, thwarted in otherwise exemplary endeavours, and unable to uncover what went wrong? And how often is an engineer who expresses doubts about the computing system he must use regarded as if he were Dante's bad blacksmith?

10. PAUSE FOR THOUGHT

Mr. Z.'s program in fig. 14 has been exciticized on several grounds. It is alleged that, since X must be uncertain by an ulp or two, the difference (X−1.0) can contain no significant figures when X is very close to 1.0, and this is why the program deserves to fail. Similarly, the expression (X. EQ. 1.0) is sinful. But such an argument has two flaws.

First, there is little significance in the number of "correct" significant figures in a calculation's intermediate results. Matrix calculations frequently generate intermediate results among which are numbers agreeing in not one figure with what would have been generated in the absence of roundoff, but the answer at the end is correct! Another example is provided by solving the differential equation

$$a\ddot{y} - 2b\dot{y} + cy = 0, \quad \text{given} \quad y(0) = y_0 \quad \text{and} \quad \dot{y}(0) = \dot{y}_0,$$

$(\dot{y} \equiv dy/dt)$ in terms of the roots $r_\pm$ of the quadratic

$$ax^2 - 2bx + c = 0.$$

If the roots are real and distinct the solution is

$$y(t) = \left(y_0 \cosh ut + (\dot{y}_0 - vy_0)\frac{\sinh ut}{u}\right)\exp vt$$

where $u \equiv (r_+ - r_-)/2$ and $v \equiv (r_+ + r_-)/2$; if the roots are coincident at r the solution is

$$y(t) = \{y_0 + (\dot{y}_0 - ry_0)t\}\exp rt;$$

* This program was distributed to other IBM 7090/7094 users via the SHARE organization in June 1964; the relevant SDA numbers are 3190, 3191 and 3192. Logarithm and exponential subroutines of comparable quality, coded by Hirondo Kuki, are now part of the Fortran libraries, distributed with IBM 7094 and System/360 machines; see also Kuki and Ascoly [7] and references cited therein, and [20].

if the roots $r_\pm = \upsilon \pm iw$ are complex

$$y(t) = \left(y_0 \cos wt + (\dot{y}_0 - \upsilon y_0) \frac{\sin wt}{w} \right) \exp \upsilon t \, .$$

For modest* values of t the solution $y(t)$ is a well-behaved function of a, b and c even though the intermediate results, namely the roots $r_\pm$, may be extremely sensitive to small changes in those coefficients, as we have seen. But the roots do not vary capriciously. If we were to alter arbitrarily those digits of the computed roots which differ from what would have been obtained in the absence of round-off, as we could if we regarded those digits as "wrong", we would do as much damage to the value of $y(t)$ computed from those altered roots as if instead we had altered the same number of terminal digits in the coefficients; in other words, we could capriciously squander half the digits carried. If those "insignificant" digits are carried in the usual way, the value of $y(t)$ computed from them will be quite satisfactory.

* This restriction is imposed because

$$\lim_{t \to \infty} y(t)$$

may be a violently discontinuous function of a, b, c, y_0 and $\dot{y}_0$.

Now we see the advantage in a subprogram which computes accurately the roots of a quadratic equation as given even when its coefficients are uncertain to an extent which may compromise half the figures in the roots. Besides shielding its user from unproductive thought, such a subprogram will preserve relationships implied by possible correlations among the errors in the coefficients; such a subprogram cannot be the weakest link in a chain of subprograms.

The second flaw in the allegation criticized above appears when the allegation is cited in support of certain hardware designs, like the CDC 6400's, which neglect to carry guard digits for addition and subtraction. We have seen what happened to the expression (1.0–0.9999 9999); now look at figs. 15a and 15b, which were produced by our 6400 using binary floating point arithmetic with "48 significant bits". As I runs from 1 to 100, something bizarre happens for $2 \leqslant I \leqslant 48$ and $I = 97$, despite the fact that arithmetic on the machine is provably monotonic.

The problem revealed in figs. 15a and 15b could be solved in any one of four ways. First, change the compiler to effect a floating point comparison (X .EQ. Y) by using only integer arithmetic manipulations; but this would occasionally malfunction when X and Y are very different (recall fig. 11) and would

```
      PROGRAM NAUGHTY (INPUT,OUTPUT,TTYOUT,TAPE1=TTYOUT)
      X = 0.5
      F = (X - 0.5**48) + X
      DO 2   I = 1, 100
       X = X*2.0
       Y = X*F
       IF( X .EQ. Y  .AND.  (X-1.) .NE. (Y-1.) )  WRITE (1,1) I
1     FORMAT(*   WHEN  I = *, I3, * ,  X .EQ. Y  BUT  X-1 .NE. Y-1 *)
2      CONTINUE
      STOP
      END
```

```
BEGIN EXECUTION      NAUGHTY
    WHEN  I =   2 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =   3 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =   4 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =   5 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =   6 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =   7 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =   8 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  41 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  42 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  43 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  44 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  45 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  46 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  47 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  48 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
    WHEN  I =  97 ,  X .EQ. Y  BUT  X-1 .NE. Y-1
STOP        NAUGHTY
>
```

Fig. 15a. How can I determine when $X = Y$ but $X - 1 \neq Y - 1$?

```
      PROGRAM NAUGHTY (INPUT,OUTPUT,TTYOUT,TAPE1=TTYOUT)
      X = 0.5
      F = (X - 0.5**48) + X
      DO 2   I = 1, 100
       X = X*2.0
       Y = X*F
       IF( X .LE. Y  .AND.  (X-1.) .GT. (Y-1.) )  WRITE (1,1) I
1     FORMAT(*   WHEN  I = *, I3, * ,  X .LE. Y  BUT  X-1 .GT. Y-1 *)
2      CONTINUE
      STOP
      END
```

```
BEGIN EXECUTION      NAUGHTY
   WHEN  I =   2 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =   3 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =   4 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =   5 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =   6 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =   7 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =   8 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =   9 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  41 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  42 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  43 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  44 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  45 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  46 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  47 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  48 ,  X .LE. Y  BUT  X-1 .GT. Y-1
   WHEN  I =  97 ,  X .LE. Y  BUT  X-1 .GT. Y-1
STOP       NAUGHTY
>
```

Fig. 15b. How can *I* determine when $X \leqslant Y$ but $X - 1 > Y - 1$?

occasionally allow division by zero in statement 3 of fig. 14. Second, change the compiler to perform additions and subtractions properly; this would require five instructions* instead of the two now executed, at a cost of perhaps doubling their execution time. Third, change the hardware so that the pseudo-rounding RX instructions (which are rarely used now) will normalize before rounding, and then alter some software to allow advantage to be taken of this change; this could cost a few million dollars if done for all CDC 6000 series machines, but the problem would then be completely eliminated.

The fourth possibility is to change the way we think about numbers. Instead of basing numerical analysis upon fewer than a dozen axioms, we could adopt a new "number" system like that suggested by van Wijngaarden, with 32 axioms which, if not categorical, appear to be at least consistent. But if the test of a scientific advance is the extent to which it permits us to know more while obliging us to remember less, such a new number system is not an advance.

Perhaps certain computer systems could be classified as dangerously addictive hallucinatory drugs, and compulsorily labelled:

"Warning. It Has Been Determined That This Computer Is Dangerous To Your Mental Health."

If the reader runs programs on one of those computers he will not be thankful for the foregoing exposé. When one of his programs fails mysteriously because of a misplaced comma in a FORMAT statement, and when he has failed to find that flaw or any other he can imagine, he may turn to these pages to see whether one of the rare anomalies revealed above has caused his trouble. How long will he spend on that wild goose chase?

* Currently $X_1 = X_2 - X_3$ is computed via the sequences

```
FX1  X2-X3   or    RX1  X2-X3
NX1  X1            NX1  X1
```

which I would replace by

```
FX1  X2-X3
NX1  X1
DX0  X2-X3
NX0  X0
RX1  X1+X0
```

11. MORE SURPRISES

"Things are seldom what they seem,
Skim milk masquerades as cream."

H.M.S.Pinafore
Gilbert and Sullivan

Rounding error analysis may be full of surprises, but it is void of major theorems. There seem to be deep reasons why this must be so, reasons which I propose to sketch now.

Many an error analyst has tried and failed to prove theorems of the form:

"To compute XXX correct to single-precision requires that YYY be computed using ZZZ-precision arithmetic."

Perhaps the failure is inevitable, for there is some possibility that machine-independent Fortran subroutines could be written to perform arbitrarily high precision floating point arithmetic without using any but REAL variables; see Dekker [10]. We shall examine a special simple example of that notion.

Let us try to evaluate $S_N = \Sigma_1^N X_J$ where N is very large ($N \geqslant 10^6$) and each X_J is computable to nearly full single-precision as a function of J and of S_{J-1}. Such a problem arises in the course of solving ordinary differential equations by discrete methods. The program

```
      S = 0.
      DO 9 J = 1,N
    9 S = S + X(J,...)
```

actually computes

$$s_n = \Sigma_1^n (1+\xi_j) x_j \quad \text{with} \quad |\xi_j| \leqslant (1+\epsilon)^{n+1-j} - 1 \,.$$

Take $\epsilon = 10^{-6}$ (as on, say, IBM System/360 machines) and $n = N = 10^6$ to see what goes wrong here; the loss of accuracy could be worse than in the second program of fig. 12. A better program is obtained by prefacing

```
DOUBLE PRECISION S
```

to that above, thereby replacing ϵ by roughly ϵ^2 and introducing little more uncertainty to s_n than is inherited from an uncertainty of a few ulps in each $x_j = (1+\chi_j)X_j$ when each $|\chi_j| < 10\epsilon$, say. But what if double-precision is unavailable (or if ϵ represents double-precision, and triple-precision is unavailable)? Can we still compute $s_n = \Sigma_1^n (1+\xi_j) x_j$ in such a way that the quotient $|\xi_j/\epsilon|$ is bounded independently of j and n except for factors like $(1+\epsilon^2)^n$?

The answer depends upon whether single-precision addition uses a guard digit or not. If it does, the following annotated program works:

```
      S = 0.                    s_0 ≡ 0
      C = 0.                    c_0 ≡ 0
      DO 9 J = 1,N              For j = 1,2,...,n in turn
        Y = C + X(J,...)        y_j ≡ (x_j+c_{j-1})(1+η_j)
        T = S+Y                 s_j ≡ (s_{j-1}+y_j)(1+τ_j)
        C = (S-T)+Y             c_j ≡ ((s_{j-1}-s_j)(1+σ_j)+y_j)(1+γ_j)
    9   S = T
      SUM = S+C (slightly better than S)
                                s_n + c_n = Σ_1^n (1+ξ_j)x_j
```

Provided $|\eta_j| \leqslant \epsilon$, $|\tau_j| \leqslant \epsilon$, $|\sigma_j| \leqslant \epsilon$ and $|\gamma_j| \leqslant \epsilon$, it may be shown that

$$1 + \epsilon_j = (1+\eta_j)\{1-\sigma_j+O((n+1-j)\epsilon^2\} \,.$$

I published this program (unannotated) in 1965 [11]. A similar program has been presented by Babuška [12], and a more complicated one by Møller [13] is further discussed by Knuth [14], pp. 201–4, from a different point of view. Similarly motivated algorithms continue to be developed; see Thompson [15].

When the program above was first published it was accompanied by a warning not to use it on machines that chopped or rounded before normalizing, as does our CDC 6400. The warning was issued with systems of differential equations in mind, but another potential application denied to that program on our machine was discovered unwittingly by van Reeken [16], who wished to compute running averages

$$A_N \equiv S_N/N$$
$$= S_{N-1} + (X_N - S_{N-1})/N$$

from the last formula. He claimed that "addition using Kahan's trick will give an error-free answer" even on machines which truncate before normalizing. He was almost right; fig. 16 exhibits an extremely rare counter-example which he could not reasonably have been expected to uncover in his tests.

There is a theorem by Viten'ko [17] which almost implies that our objective, to bound $|\xi_j/\epsilon|$ independently of j and n except for terms $O(n\epsilon^2)$, is impossible on those machines which respond, as do those which chop first and normalize later, to the statement

```
      PROGRAM BUNGLE (INPUT,OUTPUT,TTYOUT,TAPE1=TTYOUT)
C     THIS PROGRAM COMPUTES THE AVERAGE  A  OF 3000000 VALUES  X(N) ,
C     EACH BETWEEN 0.5 AND 1.5 , IN TWO DIFFERENT WAYS. ONE OF THOSE
C     WAYS USES, INSTEAD OF DOUBLE PRECISION, A TRICK WHICH ALWAYS
C     WORKS ON SOME MACHINES AND ALMOST ALWAYS WORKS ON ALL OTHERS.
C     A RARE SET OF VALUES  X(N)  FOR WHICH THE TRICK FAILS ON THE
C     CDC 6400 IS COMPUTED BY THIS PROGRAM.
      DOUBLE PRECISION  S
      REAL  N
      E = 0.5**48
      F = 2.0*E
C     THE FOREGOING CONSTANTS ARE CHARACTERISTIC OF
C       THE   CDC 6400 .
      C = 0.0
      Z = (1.0-F)+E
      N = 0.0
      S = 0.0
      A = 0.0
      DO 3   L = 1, 10
      DO 3   K = 1, 100000
      DO 3   J = 1, 3
        N = N+1.0
C     COMPUTE  X(N) .
        X = Z
        IF( L .EQ. 1 .AND. K .EQ. 1 )  GO TO 2
        IF( J .EQ. 1 )   X = 1.0+F*(N-1.0)
        IF( J .EQ. 2 )   X = 1.0-F*N
        IF( J .LT. 3 )  GO TO 2
           X = (1.0-F*N)+E*N
1       IF(((X-A)/N+C)+A .GT. A )  GO TO 2
        X = X+E
        GO TO 1
C     NOW  X  IS DETERMINED. NEXT UPDATE THE AVERAGE  A  .
2     DA = (X-A)/N + C
        T = A+DA
        C = (A-T) + DA
        A = T
        S = S+X
3     CONTINUE
      AV = S/N
      WRITE(1,9) N, AV, A
9     FORMAT( 2X,*N =*,F9.0,5X,*AV =*,F19.15,5X,*A =*,F19.15 /
     * *  NO. OF ITEMS*,8X,*TRUE AVERAGE*,13X,*COMPUTED AVERAGE*)
      STOP
      END
:

BEGIN EXECUTION      BUNGLE
  N = 3000000.      AV =    .999999998223636      A =    .999999999999996
  NO. OF ITEMS        TRUE AVERAGE               COMPUTED AVERAGE
STOP       BUNGLE
>
```

Fig. 16. An egrègious average.

$$B = C + D$$

by computing $b = (1+\gamma)c + (1+\delta)d$ with $|\gamma| \leqslant \epsilon$ and $|\delta| \leqslant \epsilon$. Viten'ko showed that the best that could be done when, say, $N = 8$ was to compute the expression

$$((X_1+X_2)+(X_3+X_4)) + ((X_5+X_6)+(X_7+X_8))$$

which, in general, would allow $|\xi_j/\epsilon|$ to grow as fast as $\log_2 N$. But his hypotheses do not take account of all that is known about γ and δ. Consequently, the program annotated above may be made to work on all major North American computers with floating point hardware by replacing the statement

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

with

```
F = 0
IF(SIGN(1.,Y) .EQ. SIGN(1.,S)) F = (0.46*T-T)
   + T
C = ((S-F)-(T-F)) + Y
```

This is not the place to explain why the modified program works on all such machines, nor why the magic number 0.46 was chosen. Rather, the reader should observe that programs may work, on some machines, far better than he can prove. Next consider

a programmer faced with the task of producing a program which works well and can be proved to work well. He also faces a dilemma; should he try to prove that a simple program on hand works well, or should he write another more complicated program more amenable to proof? On some machines the dilemma is acute.

That tricky programs like those above contain surprises is not surprising, but sometimes surprises are well hidden. For instance, consider the solution of a cubic equation

$$Q(x) \equiv a_0x^3 + 3a_1x^2 + 3a_2x + a_3 = 0 .$$

If its coefficients are in error by as much as one ulp its roots may be accurate to only $\frac{1}{3}$-precision, as is exemplified by

$$x^3 - 3x^2 + 3x - (1-\epsilon) = 0$$

whose roots are the three values of $1 + \epsilon^{1/3}$. Any algorithm for solving a cubic will encounter roundoff which can, in part at least, be regarded as perturbing the coefficients; see Wilkinson [18]. Although he definitely does not say so, reading his book might give the impression that triple-precision arithmetic will be needed to get the roots to single-precision. Of course the critical cubics, those with three nearly coincident roots, can be transformed, by a linear substitution which moves the origin nearer to the roots, into a less delicate condition; but G.W.Stewart [19] shows that the usual way of effecting such a transformation does not avoid the damaging perturbations. Nevertheless, my 1968 notes [20] contain a different form of the transformation which avoids the worst of the perturbations;

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

when

$$b_0 \equiv a_0 \quad b_1 \equiv a_0c + a_1 \quad b_2' \equiv a_1c + a_2 \quad b_3' \equiv a_2c + a_3$$

$$b_2 = b_1c + b_2' \quad b_3'' \equiv b_2'c + b_3'$$

$$b_3 \equiv b_2c + b_3'' .$$

Given single-precision coefficients a_i and a suitable single-precision c, this transformation is to be carried out using double-precision arithmetic. The choice of c can be effected in an innocent machine-independent fashion. The final result is a program which accepts single-precision coefficients, uses double-precision arithmetic, and produces roots correct to nearly single-precision, as if triple-precision arithmetic had been used. The program works on all major North American machines; to prove that it works, one must acknowledge that catastropic cancellation can be a good thing.

12. ESCAPE FROM ROUNDING ERROR ANALYSIS

There are three ways to escape rounding error analysis without abandoning computation. One is to use multi-precision arithmetic so precise that errors are "obviously" negligible if they occur at all. A second way is to use well implemented Interval Arithmetic. Since Moore [21], Hansen [22], Nickel [23], I [20] and others have written extensively about Interval Arithmetic, little is left to say about it here beyond this; no other development in computer systems would assist engineers and others like them to do numerical computations more safely then would the appearance of Interval Arithmetic as universally accessible in Fortran as are double-precision and complex arithmetic. For example, by using 4-significant decimal Interval Arithmetic we obtain almost effortlessly the estimates

$$R_+ \in [.9987, 1.020] \quad , \quad R_- \in [.9781, .9988]$$

for the roots of fig. 3's quadratic provided those roots are real, and

$$R_r \in [.9987, .9988] \quad , \quad R_i \in [0, 0.02105]$$

for the roots $R_r \pm R_i$ if they are complex. More important, if all we know about the coefficients is, say,

$$A \in [47.46, 47.56] \ , \quad B \in [47.40, 47.50] \ ,$$

$$C \in [47.34, 47.44]$$

then the inferences

$$R_+ \in [.9756, 1.071] \quad , \quad R_- \in [.9315, 1.001] \quad \text{or}$$

$$R_r \in [.9966, 1.001] \quad , \quad R_i \in [0, 0.06990]$$

(which are nearly unimprovable) come more economically, by far, from a direct application of Interval Arithmetic than from any other scheme. The fact that Interval Arithmetic can be abused, and then will give

wretchedly pessimistic error bounds, is no excuse to deny its use to the computer using public. I suspect that Interval Arithmetic is still so little used mainly because deficiencies in some current floating point hardware designs metamorphose into embarrassing inefficiencies when Interval Arithmetic is implemented. Even so, Interval Arithmetic tends to be cheaper than the human labour it supplants.

The third way to escape is to realize that there are other kinds of errors than rounding errors. Errors in data and errors in intentional approximations to mathematical relationships cannot be dispelled by the means described above, and are therefore the preferred preoccupation of error analysts. I shall give two examples drawn from my own work.

13. TRAJECTORY PROBLEMS

"I shot an arrow in the air,
It fell to earth, I know not where."

The Arrow and the Song
Longfellow

Consider a system of n ordinary differential equations

$$\dot{y} = f(y,t) + r(t)\,, \quad y(0) = y_0 + w_0$$

in which uncertainties are represented by n-vectors $r(t)$ and w_0 about which we know only bounds like

$$\Omega_0 \geqslant \|w_0\| \quad \text{and} \quad \rho(t) \geqslant \|r(t)\| \quad \text{for} \quad t \geqslant 0\,.$$

Our object is to compute a bound

$$\Omega(t) \geqslant \|y(t) - z(t)\|$$

for the difference between the uncertain solution vector $y(t)$ and the unperturbed solution $z(t)$ of

$$\dot{z} = f(z,t)\,, \quad z(0) = y_0\,.$$

The source of the uncertainty $r(t)$ is not important here. It could arise from the numerical method used to solve $y(t)$'s differential equation, with $z(t)$ representing what the numerical method produces (see N.F.Stewart [24]). Alternatively, $r(t)$ could represent unknown but bounded perturbing forces acting upon a physical system $y(t)$ whose unperturbed motion would be $z(t)$. Most likely both sources of error would contribute to $r(t)$, as they would to w_0.

Over the past century several methods have been proposed for computing $\Omega(t)$; significant contributions have been made recently by Moore [21] and Krückeberg [25]. But *all* methods described so far share an outstanding defect; they tend to produce a function $\Omega(t)$ which grows, as $t \to \infty$, exponentially faster than $\|y(t) - z(t)\|$ can grow, even when the differential equation is linear, and in most cases even when it is linear with constant coefficients chosen in an unlucky way (see L.W.Jackson [26, 27]). There is one exception.

In 1966 I proposed [28] that ellipsoids be used to produce $\Omega(t)$. The idea was to compute a positive definite $n \times n$ matrix $A(t)$, the solution of an auxiliary system of differential equations solved simultaneously with $z(t)$'s equation, which would represent an ellipsoid $\mathrm{A}(t)$ as follows:

$$x \in \mathrm{A} \quad \text{if and only if} \quad x' A^{-1} x \leqslant 1\,.$$

$A(t)$'s differential equation was to be so chosen that $y(t) - z(t) \in \mathrm{A}(t)$ for all $t \geqslant 0$. The scheme will be described below simply for linear differential equations although it works on non-linear equations too, until $\mathrm{A}(t)$ becomes so large as to grow spuriously and unavoidably too fast.

Let $w(t) \equiv y(t) - z(t)$, and assume

$$\dot{w} = Jw + v\,, \quad w(0) = w_0$$

where $J(t)$ is a known $n \times n$ matrix but no more is known about $v(t)$ and w_0 than two ellipsoids $\mathrm{V}(t)$ and A_0 such that

$$w_0 \in \mathrm{A}_0 \quad \text{and} \quad v(t) \in \mathrm{V}(t) \quad \text{for} \quad t \geqslant 0\,.$$

In other words we assume positive definite matrices A_0 and $V(t)$ are given such that $w_0 A_0^{-1} w_0 \leqslant 1$ and $v' V^{-1} v \leqslant 1$ for all $t \geqslant 0$. For example, given $\rho^2 \geqslant v'v$ for all $t \geqslant 0$ we should set $V = \rho^{-2}$. Now let $\mathrm{W}(t)$ denote the "reachable set" of all solutions $w(t)$ obtained by letting w_0 and $v(t)$ range over the sets A_0 and $\mathrm{V}(t)$ respectively. In general $\mathrm{W}(t)$ is not an ellipsoid; we seek $\mathrm{A}(t) \supseteq \mathrm{W}(t)$ for all $t \geqslant 0$.

THEOREM. If $A(t)$ satisfies*

$$\dot{A} \geqslant JA + AJ' + \gamma A + V/\gamma\,, \quad A(0) \geqslant A_0$$

* Writing "$X \geqslant Y$" for symmetric matrices means that $X - Y$ is positive semi-definite; $x'(X-Y)x \geqslant 0$ for all x.

for any $\gamma(t) > 0$ and for all $t \geqslant 0$ then $A(t)$ represents an ellipsoid $\mathrm{A}(t) \supseteq \mathrm{W}(t)$.

To apply this theorem we might replace its first two $\geqslant$ signs by = signs and solve the resulting differential equation numerically for $A(t)$ simultaneously with the calculation of, say, $z(t)$, provided we knew how to choose $\gamma(t)$. There are many reasonable choices available. For example see the following.

Corollary. If V is constant and $A(0) = A_0 = 0$ and $J(t)$ is bounded for all $t \geqslant 0$ and $\dot{A} = JA + AJ' + \gamma A + V/\gamma$ with $\gamma(t) \equiv 1/t$ then $\mathrm{W}(t) \subseteq \mathrm{A}(t) \subseteq \lambda(t)\mathrm{W}(t)$ where $\lambda(t)/\sqrt{1+t}$ is bounded for all $t \geqslant 0$.

In other words, here is a case where the error bound cannot over-estimate the possible error by more than a bounded multiple of $\sqrt{1+t}$. There are many other cases of considerable practical importance where $\gamma(t)$ can so be chosen that the error bound will never grow arbitrarily larger than the possible error. For example, if w's differential equations are the variational equations for the equations of motion of a satellite in orbit about a lumpy central body whose gravitational field deviates slightly from the inverse-square law in an unknown but bounded way, or if the equations of motion concern a pendulum swinging in a draft of gas of unknown but small and bounded density and velocity, $\gamma(t)$ can easily so be chosen that the ellipsoid $\mathrm{A}(t)$ will grow at the same rate as the reachable set $\mathrm{W}(t)$ for all $t \geqslant 0$ until $\mathrm{A}(t)$ becomes so large that nonlinearities in the equations of motion dominate its growth. Calculations, some performed with the aid of a particularly convenient program written by Gabel [29] to solve differential equations automatically on the 7094, have borne out these claims. Details must appear elsewhere.

14. ILL-POSED PROBLEMS

> My object all sublime –
> I shall achieve in time –
> To let the punishment fit the crime.
>
> *Mikado*
> W.S.Gilbert

Among the most perplexing numerical computations are those whose results, though intended to mimic an ostensibly well-behaved physical configuration, turn out ill-behaved. Are they ill-behaved merely because the numerical computation was performed ineptly? Or is the physical system not so well-behaved as was presumed? Or does its mathematical model contain a flaw, *not* a mistake, which condemns every straight-forward numerical method to confusion? This last possibility can arise in two ways. On the one hand, intermediate variables may have been introduced which are occasionally redundant, thereby allowing partly arbitrary and possibly unbounded numerical values to intrude enormous rounding errors into the computation. On the other hand, the physical system may obey precisely laws which can only be approximated numerically; the small errors so introduced may then correspond to physically impossible perturbations with physically impossible consequences.

To what extent can the foregoing three questions be resolved by numerical means alone without descending to numerological augury? We wish not to re-formulate a new mathematical model unless we have to, and then not until we know what is wrong with the old model. We hope to avoid the kind of deft and inspired analysis exemplified by, say, Dorr [30] and Babuška [31], since that may well lie beyond our talents.

Error analysis offers a resolution based upon two notions. First, the uncertainty attributed to data is itself a datum. Secondly, when experimental observations are subjected to computational processing, the program becomes a part of the experimental apparatus, and subject to the same scientific criteria concerning the reproducibility of meaningful results in the face of ostensibly negligible variations. These notions will be illustrated by application to a simple linear least squares problem.

Given an $m \times n$ matrix F with $m \geqslant n$, and an m-vector g, we seek that n-vector x which minimizes $\|g - Fx\|$; when the minimizing x is not unique (i.e., when the columns of F are linearly dependent) we further stipulate that, say, $\|x\|$ should be minimized. The vector norm used here is $\|z\| \equiv \sqrt{z'z}$, and we shall use the natural matrix norm $\|\mathrm{Z}\| \equiv \max_z \|\mathrm{Z}z\|/\|z\|$ although any other orthogonally invariant matrix norm could be adapted to our purposes. The minimizing vector x turns out to be $F^\dagger g$ where the pseudo-inverse $F^\dagger$ is uniquely defined formally by the familiar equations

$$FF^\dagger F = F\,, \quad F^\dagger FF^\dagger = F^\dagger\,, \quad (F^\dagger F)' = F^\dagger F\,,$$

$$(FF^\dagger)' = FF^\dagger\,.$$

When F has full column-rank n, $F^\dagger = (F'F)^{-1}F'$.

The literature abounds with methods for computing $F^\dagger$ and $F^\dagger g$. Some of the best are explained by

Golub and his collaborators; see the several references. Certain cases when F is of full rank but badly ill-conditioned ($\|F^\dagger\|\|F\|$ is huge) are discussed nicely by Gautschi [32,33] and by Wilson [34]. Another special case in which we seek to choose Δg, subject to a given constraint $\gamma \geqslant \|\Delta g\|$, to nearly minimize $\|F^\dagger(g+\Delta g)\|$ when F is badly ill-conditioned is discussed by Miller [35] and mentioned by Golub and Kahan [36]. But nobody has considered what to do when F is uncertain, although this matter is touched obliquely by G.W.Stewart [24] and by Pereyra [37].

We shall consider the implications of uncertainty in F for the computation of $F^\dagger$. Specifically, given a tolerance $\phi > 0$ such that all $F + \Delta F$ with $\|\Delta F\| \leqslant \phi$ must be regarded as indistinguishable for practical purposes, what should be done when $(F+\Delta F)^\dagger$ is found to vary violently discontinuously as ΔF ranges over the allowed set?

First some apparatus is needed. Let the n singular values of F be denoted in order by

$$\phi_1 \geqslant \phi_2 \geqslant \cdots \geqslant \phi_n \geqslant 0 .$$

These may be computed at modest cost by methods described in Golub [38] and in Golub and Reinsch [39]. Note that the singular values of $F^\dagger$ are the reordered numbers $\phi_j^\dagger$, where $\phi^\dagger \equiv 1/\phi$, except $0^\dagger \equiv 0$. According to Mirsky ([41], theorem 2) for $k = 1, 2, \ldots, n$

$$\phi_k = \min \|\Delta F\| \quad \text{over} \quad \operatorname{rank}(F+\Delta F) < k .$$

Consequently no singular value of $F + \Delta F$ can differ from the correspondingly numbered singular value of F by more than $\|\Delta F\|$; and just as $\phi_1 = \|F\|$ so is

$$\|F^\dagger\| = 1/\min \|\Delta F\| \quad \text{over} \quad \operatorname{rank}(F+\Delta F) < \operatorname{rank}(F) .$$

Finally, the following little known but easily verified and useful formula,

$$E^\dagger - F^\dagger = -F^\dagger(E-F)E^\dagger + (1-F^\dagger F)(E-F)'E^{\dagger\prime}E^\dagger + F^\dagger F^{\dagger\prime}(E-F)'(1-EE^\dagger) ,$$

has as a corollary

$$\|E^\dagger - F^\dagger\| \leqslant \|E-F\| \sqrt{\frac{\|E^\dagger\|^5 - \|F^\dagger\|^5}{\|E\| - \|F\|}}$$

$$\leqslant \sqrt{5}\|E-F\| \max(\|E^\dagger\|, \|F^\dagger\|)^2 .$$

The foregoing apparatus is the justification for the following assertions.

The first step is to exhibit $F = P\Lambda Q$ where P and Q are orthogonal matrices ($P'P = Q'Q = QQ' = 1$) and $\Lambda = \operatorname{diag}(\phi_1, \phi_2, \ldots, \phi_n)$; this can be done by methods mentioned above. Next compare the tolerance ϕ with the singular values ϕ_j. If $\phi \ll \phi_n$ then for all $\|\Delta F\| \leqslant \phi$

$$\|(F+\Delta F)^\dagger\| \leqslant 1/(\phi_n - \phi)$$

and

$$\|(F+\Delta F)^\dagger - F^\dagger\| \leqslant (1+\phi_n^2/(\phi_n-\phi)^2)^{1/2}\phi/\phi_n^2 ;$$

thus, we have a bound for the change in $F^\dagger$ caused when F is changed by no more in norm than the tolerance ϕ.

The interesting case occurs when $\phi_k \geqslant \phi > \phi_{k+1}$ for some $k < n$; this means that among the matrices $F + \Delta F$ with $\|\Delta F\| \leqslant \phi$ are some of rank k, $k+1$, ... and n. Every time $F + \Delta F$ changes rank, $(F+\Delta F)^\dagger$ jumps infinitely violently. Clearly the least squares problem is now ill-posed because a matrix $F + \Delta F$ indistinguishable from F has only k linearly independent columns. The last $n - k$ rows of Q exhibit the independent linear combinations of the columns of F which nearly vanish. As $F + \Delta F$ runs through matrices of minimal rank k with $\|\Delta F\| \leqslant \phi$, $(F+\Delta F)^\dagger$ varies continuously and differs by no more than $\sqrt{5}(\phi+\phi_{k+1})/(\phi_k-\phi)^2$ in norm from a computable distinguished choice

$$(F+\hat{\Delta}F)^\dagger \equiv Q' \operatorname{diag}\{\phi_1^{-1}\phi_2^{-1}, \ldots, \phi_k^{-1}, 0, 0, \ldots, 0\}P' .$$

The corresponding $\hat{x} \equiv (F+\hat{\Delta}F)^\dagger g$ has the property that it, like $(F+\hat{\Delta}F)$, is a continuous function of the data F and g for variations small compared with $\phi_k - \phi$. Finally, $\|g - F\hat{x}\|$ may be rather larger than minimal, but if so it cannot be reduced without replacing $\hat{x}$ by a drastically larger vector x which must change violently when F is changed negligibly. In other words, $(F+\hat{\Delta}F)^\dagger$ reveals something about the data F, g which is independent of allegedly negligible (smaller than ϕ) variations in the data. In this respect, an ill-posed problem has been replaced usefully by a well-posed one, and by numerical means alone. When neither condition $\phi \ll \phi_n$ nor $\phi_k \gg \phi > \phi_{k+1}$ is satisfied, i.e., when ϕ is not much smaller than the next larger singular value, the given least squares problem must be regarded as intrinsically ill-posed in a way that will not yield to numerical methods alone.

ACKNOWLEDGEMENTS

This paper is distilled from a fraction of the works of so many people that, were I to try to acknowledge as many as could be named in these pages, I could only injure more of them by omission. Better so to injure almost all. Besides, their names are familiar to anyone who reads the journals of numerical analysis and computing. But I am indebted for many acts of kindness to G.E.Forsythe, to A.S.Householder, to J.H.Wilkinson, to my former colleagues and teachers at the University of Toronto, Canada, and to my colleagues now at the University of California at Berkeley.

The work described here has been supported in part by grants from the National Research Council of Canada and from the U.S. Office of Naval Research.

REFERENCES

[1] B.T.Smith, Error bounds for zeros of a polynomial based upon Gerschgorin's theorem, JACM 17 (1970) 661–674.

[2] W.Börsch-Supan, Residuenabschätzung für Polynom-Nullstellen mittels Lagrange-Interpolation, Numer. Math. 14 (1970) 287–296.

[3] D.A.Adams, A stopping criterion for polynomial root finding, Comm. ACM 10 (1967) 655–658.

[4] A.M. Ostrowski, Solution of Equations and Systems of Equations, 2nd ed. Acad. Press, New York (1966).

[5] W.Kahan, 7094-II System support for numerical analysis, SHARE Secretarial Distribution SSD-159, item C4537 (166), (An amended version is reprinted in my 1968 notes.).

[6] W.J.Cody, Jr., Software for the elementary functions, presented at the Mathematical Software Symposium at Purdue University (April 1970).

[7] H.Kuki and J.Ascoly, Fortran extended-precision library, IBM Syst. J. 10 (1971) 39–61.

[8] H.E.Fettis and J.C.Caslin, Errata in tables of toroidal harmonics, Math. of Comp. 25 (1971) 405–408.

[9] R.R.Korfhage, On a sequence of prime numbers, Bull. Amer. Math. Soc. 70 (1964) 341–342, retracted on p. 747.

[10] T.J.Dekker, A floating-point technique for extending the available precision, report MR 118/70, Mathematisch Centrum, Amsterdam (1970).

[11] W.Kahan, Further remarks on reducing truncation errors, Comm. ACM 8 (1965) 40.

[12] I.Babuška, Numerical stability in mathematical analysis, in: Proc. IFIP Congress (1968) vol. 1, A.J.H.Morrell, ed, (North-Holland, Amsterdam).

[13] O.Møller, Quasi double-precision in floating point addition, BIT 5 (1965) 37–50 and 251–255.

[14] D.E.Knuth, The Art of Computer Programming, vol. 2 (1969) Semi-numerical Algorithms, Addison-Wesley, Massachusetts.

[15] R.J.Thompson, Improving round-off in Runge-Kutta computations with Gill's method, Comm. ACM 13 (1970) 739–740.

[16] A.J.van Reeken, Dealing with Neely's algorithms, letter to the ed., Comm. ACM 11 (1968) 149–150.

[17] I.V.Viten'ko, Optimum algorithms for adding and multiplying on computers with a floating point, USSR Computational Math. and Math. Physics 8, #5 (1968) 183–195.

[18] J.H.Wilkinson, Rounding errors in algebraic processes, Nat'l Phys. Lab. Notes on Applied Science, no. 32 (1963) HMSO, London.

[19] G.W.Stewart III, On the continuity of the generalized inverse, SIAM J. Appl. Math. 17 (1969) 33–45.
G.W.Stewart III, Error analysis of the algorithm for shifting the zeros of a polynomial by synthetic division, Math. of Comp. 25 (1971) 135–139.

[20] W.Kahan, Error in numerical computation, part of the notes for summer course #6818, Numerical Analysis, University of Michigan Engineering Summer Conferences, Ann Arbor, Michigan (1968).

[21] R.E.Moore, Interval Analysis (Prentice-Hall, New Jersey 1969).

[22] E.R.Hansen, ed., Topics in Interval Analysis, (Oxford U.P., London, 1969).

[23] K.Nickel, Error-bounds and computer arithmetic (1969) 54–62, in: Information Processing 1968, Proc. IFIP Congress 1968, vol. 1, A.J.H.Morrell ed. (North-Holland, Amsterdam).

[24] N.F.Stewart, Certain equivalent requirements of approximate solutions of $x = f(t,x)$, SIAM J. Numer. Anal. 7 (1970) 256–270.

[25] F.Krückeberg, Ordinary differential equations,(1969) 91–97, in: E.R.Hansen's Topics in Interval Analysis.

[26] L.W.Jackson, A comparison of ellipsoidal and interval arithmetic error bounds, abstract alone in SIAM Rev. 11 (1969) 114.

[27] L.W.Jackson, Automatic error analysis for the solution of ordinary differential equations, Ph.D. Thesis/Tech. Rep. no. 28 (1971) Computer Science Dept., Univ. of Toronto.

[28] W.Kahan, A computable error-bound for systems of ordinary differential equations, abstract alone in SIAM Rev. 8 (1966) 568–569.

[29] G.F.Gabel, A predictor-corrector method using divided differences, M.Sc. Thesis/Tech. Report No. 5, Computer Science Dept., Univ. of Toronto (1968).

[30] F.W.Dorr, An example of ill-conditioning in the numerical solution of singular perturbation problems, Math. of Comp. 25 (1971) 271–283.

[31] I.Babuška, Numerical stability in problems in linear algebra, Tech. Note BN-663, Inst. Fluid Dynamics and Appl. Math., Univ. of Maryland, College Pk., Md. 20742 (1970).

[32] W.Gautschi, Construction of Gauss-Christoffel quadrature formulas, Math. of Comp. 22 (1968) 251–270.

[33] W.Gautschi, On the construction of Gaussian quadrature rules from modified moments, Math. of Comp. 24 (1970) 245–260.

[34] M.Wayne Wilson, Discrete least squares and quadrature formulas, Math. of Comp. 24 (1970) 271–282.

[35] K.Miller, Least squares methods for ill-posed problems with a prescribed bound, SIAM J. Math. Anal. 1 (1970) 52–74.

[36] G.H.Golub and W.Kahan, Calculating the singular values and pseudo-inverse of a matrix, J. SIAM Numer. Anal. (B) 2 (1965) 205–224;
G.H.Golub and J.H.Wilkinson, Note on the iterative refinement of least squares solutions, Numerische Math. 9 (1966) 139–148.

[37] V.Pereyra, Stability of general systems of linear equations, Aequationes Math. 2 (1969) 194–206.

[38] G.H.Golub, Least squares, Singular values and matrix approximations, Aplikace Matematiky 13 (1968) 44–51.
G.H.Golub, Matrix decompositions and statistical calculations, in: Statistical Computation Acad. Press, New York (1969) 365–397.

[39] G.H.Golub and C.Reinsch, Singular value decomposition and least squares solutions, Numer. Math. 14 (1970) 403–420.

[40] D.B.DeLury, Computations with approximate numbers, The Mathematics Teacher 51 (1958) 521–530.

[41] L.Mirsky, Symmetric gauge functions and unitarily invariant norms, Quart. J. Math. (2nd series) 11 (1960) 50–59.