RATIONAL ARITHMETIC in FLOATING-POINT

W. Kahan
September 20, 1986

Abstract : Calculating $M/N := A/B \pm C/D$ in lowest terms, given the integers A, B, C and D, is a task taught in Elementary schools; and it is an easy exercise in Computer Programming too provided the given integers must be less than half as wide as the widest integers that can be handled conveniently by the computer's hardware or by its programming language. But that program becomes much more complicated (and slower) if it is naively expected to perform correctly whenever all six of our integers A, B, C, D, M and N are allowed to grow almost as wide as those widest convenient integers. This simple task illustrates why the art of programming entails sometimes a delicate balance between, on the one hand, the simplicity and aesthetic appeal of the specifications and, on the other hand, the complexity and efficiency of the implementation.

Introduction:

The obvious way to calculate

$$M/N := A/B \pm C/D \quad \text{in lowest terms}$$

is to first calculate

$$M \times k := A \times D \pm B \times C \text{ and } N \times k := B \times D$$

and then divide them by their Greatest Common Divisor

$$k := \gcd(M \times k, N \times k).$$

But the obvious way is no way to calculate

$$31/1897\,51872 = 1234\,56799/123456 - 9882\,97396/988291$$

on a calculator that carries only ten significant decimals because first

$$\begin{aligned} M \times k &:= 1234\,56799 \times 988291 - 123456 \times 9882\,97396 \\ &= 12201\,12433\,40509 - 12201\,12433\,20576 \\ &= 19933 \end{aligned}$$

and

$$N \times k := 123456 \times 988291 = 12\,20104\,53696$$

would have to be calculated in order to reveal

$$k := \gcd(19933, 12\,20104\,53696) = 643.$$

On that calculator, the two fifteen-digit products would both round to the same value (12201 12433 00000) to ten significant digits, yielding zero for $M \times k$; and $N \times k$ would get

rounded off too. However, because the desired final results $M = 31$ and $N = 1897\ 51872$ can be held exactly in that calculator, a way to compute them exactly ought to exist. An algorithm that does so without merely simulating arithmetic to at least fifteen digits is the subject of this note. The algorithm is not simple, but it is far simpler than simulating multi-word arithmetic in BASIC .

The Computing Environment:
There are limits to the widths of the integers and floating-point variables supported conveniently in programming languages like Fortran, BASIC, Pascal and *C*. Integers on some computers may be no wider than 16 bits, running from -32768 to 32767; on most other computers the integers occupy 32 bits, running from -21474 83648 to 21474 83647. Integers bigger than that lose their leftmost bits to *Overflow*, usually without any warning accessible to the higher-level language program. Floating-point variables, limited to 24 significant bits on some machines, to 53 or 56 on most others, can handle much bigger integers; but integers bigger than

$$\begin{aligned} 2.0^{24} &= 167\ 77216.0 \quad \text{or} \\ 2.0^{53} &= 9\ 00719\ 92547\ 40992.0 \quad \text{or} \\ 2.0^{56} &= 72\ 05759\ 40379\ 27936.0 \quad \text{respectively} \end{aligned}$$

lose their rightmost bits to *Roundoff*, and consequently become multiples of powers of 2 even when ideally they should have been odd. Similarly, on a typical ten-digit calculator, integers bigger than 1 00000 00000 get rounded off to multiples of powers of ten. Rounding errors occur without any warning to the program (except on machines that conform to IEEE standards 754 and 854, which require that rounding errors signal *Inexact*.) That lack of warning obliges programmers to clutter some programs with tests of the magnitudes of all intermediate results lest incorrect final results be produced with no indication that they are wrong.

Let Λ stand for the smallest positive integer beyond which some digit must be lost to overflow or roundoff; the previous paragraph tenders values of Λ appropriate for various machines. Λ is what is meant by "the widest integer that can be handled conveniently by the computer's hardware or by its programming language." The obvious way to calculate M/N described above would obviously work if $|\ A \times D\ |, |\ B \times C\ |$ and $|\ B \times D\ |$ were all somewhat smaller than Λ, as would surely be the case if $|\ A\ |, |\ B\ |, |\ C\ |$ and $|\ D\ |$ were all somewhat smaller than $\sqrt{\Lambda}$. The vagueness here implied by the word "somewhat" allows for sloppy implementations of floating-point arithmetic that, on some machines, introduce unnecessary rounding errors when integer results approach Λ too closely. Notwithstanding that vagueness, an algorithm will be presented that calculates M and N exactly whenever they and the given integers A, B, C and D are all somewhat smaller in magnitude than Λ rather than merely $\sqrt{\Lambda}$.

Rem, gcd, and Lowest Terms:
Our algorithm will require certain utilities which, if not already present in the programming environment, will have to be programmed from scratch. Reducing $(M \times k)/(N \times k)$ to its lowest terms M/N requires that $k = \gcd(M \times k, N \times k)$ be computed; and the fastest ways to compute gcd's require that remainders be computed. Let

$$\text{rem}(x, y) := x - y \times (\text{the integer nearest } x/y) \text{ provided } y \neq 0.$$

This is consistent with the definition of the operation rem that must be present in programming environments that conform to the IEEE standards 754 and 854 for floating-point arithmetic. In other environments, rem must be composed from other primitives. In Fortran the *generic intrinsic* function MOD serves to define REM thus:

```
GENERIC FUNCTION REM(x,y)
      REM = MOD(x,y)
      IF ( ABS(REM) .GT. ABS(y - REM) ) REM = y - REM
      RETURN
      END
```

Absent REM and MOD, the following procedure might be used:

```
function rem(x,y):
      q := x/y;
      n := q rounded to the nearest integer;
      return rem := x - y * n;
      end.
```

Both procedures can malfunction when x approaches or exceeds Λ in magnitude; the following example will show how roundoff in x/y and $y \times n$ causes trouble.

Suppose floating-point arithmetic is rounded to six significant decimals, for which $\Lambda = 1000000$. Now take $x = 999999.0$ and $y = 9901.0$, whereupon $x/y = 100.99979\ 80002\ ...$ must round to $q = 101.000$. Then $n = q$, but $y \times n = \Lambda + 1$ must round to Λ, which wrongly returns -1.0 instead of -2.0 for rem. Similar rounding errors inside the implementation of MOD can return -1.0 instead of 9899.0 for MOD(999999.0, 9901.00).

If the quotient x/y were chopped instead of rounded, no such malfunctions could occur. With rounding, they can be avoided by keeping $|\ x\ |$ and $|\ y\ |$ both smaller than $\Lambda/2$. If the error bound for floating-point division is vague, as it is for CRAYs, we can compensate for ignorance by further restricting $|\ x\ |$ and $|\ y\ |$; that is why phrases like "somewhat smaller than Λ" have been uttered above.

Having found a way to compute $\mathrm{rem}(x,y)$ well enough that

$(x - \mathrm{rem}(x,y))/y$ is an integer *exactly*, and
$|\ \mathrm{rem}(x,y)\ | \leq |\ y\ |/2$ *roughly*,

we may use it to compute Greatest Common Divisors quickly thus:

```
function gcd(x,y):
      while y ≠ 0 do { temp := y;
                       y := rem(x,y);
                       x := temp };
      return gcd :=| x |; end.
```

Besides the usual properties for positive integers x and y, namely

$\gcd(x,y)$ is the largest integer such that
$x/\gcd(x,y)$ and $y/\gcd(x,y)$ are both integers exactly,

this procedure $\gcd(x, y)$ has useful properties when its arguments are negative integers or zero;

$$\gcd(x, y) = \gcd(|\, x \,|, |\, y \,|) \text{ and } \gcd(x, 0) = \gcd(0, x) = |\, x \,| .$$

These properties simplify the explanation of the assertion

$$\text{"}M/N \text{ is in lowest terms,"}$$

which shall now be taken to mean that integers M and N satisfy

$$\text{"}N \geq 0, \text{ and either } \gcd(M, N) = 1 \text{ or } M = N = 0.\text{"}$$

We shall abbreviate "in lowest terms" to "ilt" and use it not only as an adjective but also as an operator that maps pairs of integers to pairs thus:

function $Ilt(x, y)$:
    $g :=$ copysign(max{gcd(x, y), 1}, y);
    return $Ilt := (x/g, y/g)$;
    end.

Now asserting that $(M, N) = Ilt(x, y)$ means the same thing as

$$M/N = x/y \text{ ilt.}$$

Idealized Rational Operations

The mapping Ilt provides a unique pair of integers (M, N) to represent each rational number $M/N = x/y$ ilt, including also $\pm 1/0 = \pm\infty$, as well as a representation for the entity 0/0 called "NaN" (for "Not a Number") in the IEEE standards for floating-point arithmetic. But those standards also specify how +0 and -0 will behave arithmetically in case a programmer chooses to distinguish them, something that cannot be done usefully on most machines that do not conform to those standards. Without a well-behaved signed zero, attempts to distinguish between $\pm\infty$ would run afoul of identities like $M/N = -1/(-N/M)$ when $M = 1$ and $N = 0$. That is why we shall herein regard ∞ as unsigned, like 0, as if the ends of the real axis had been lifted and joined to form a circle out of it. Rational operations consistent with that picture are defined in a familiar way as follows:

$$\begin{aligned} A/B \pm C/D &:= (A \times D \pm B \times C)/(B \times D) \text{ ilt respectively;} \\ (A/B) \times (C \times D) &:= (A \times C)/(B \times D) \text{ ilt;} \\ (A/B) \div (C/D) &:= (A \times D)/(B \times C) \text{ ilt;} \end{aligned}$$

$$A/B = C/D \text{ just when } A \times D = B \times C \text{ but } |\, A \times C \,| + |\, B \times D \,| \neq 0.$$

Thus, the set of all rational numbers, augmented by ∞ and 0/0, constitutes a system closed under the rational operations so defined. But the subset of rational numbers M/N representable conveniently on our computer, those for which $|\, M \,|$ and $|\, N \,|$ do not exceed Λ, does not constitute a closed system; instead it poses a challenge to implement the rational operations correctly for those operands and results that do lie within the subset.

Implementations of multiplication, division and equality-testing are entirely straightforward, as follows below, provided all operands are ilt. In other words, the operands are presumed to be pairs of integers that will pass unchanged through the function Ilt, and the results will do the same provided their magnitudes are somewhat smaller than Λ.

```
function Product(A,B,C,D): ... to get (A/B) × (C/D) ilt
    k := max{1,gcd(A,D)};    m := max{1,gcd(B,C)};
    return Product := ((A/k) × (C/m),(B/m) × (D/k));
    end.
```

Note that if the final results are all somewhat smaller in magnitude than Λ then the same must be true of all intermediate results $A/k, B/m, C/m$ and D/k, so the final results are right.

```
function Quotient(A,B,C,D): ... to get (A/B) × (C/D) ilt
    return Quotient := Product(A,B,D,C);
    end.
```

```
logical function Equal(A,B,C,D): ... does A/B = C/D ?
    if (B = 0 and D = 0) then { if (A = 0 or C = 0) then Equal := FALSE
                                                    else Equal := TRUE }
                         else { if (A = C and B = D) then Equal := TRUE
                                                    else Equal := FALSE };
    return Equal;                                   end.
```

This procedure *Equal* depends crucially upon the presumption that its arguments A/B and C/D are ilt. Note also that $0/0$ is not equal to anything, not even itself, since it's "Not a Number."

Addition and subtraction are complicated procedures because they have to cope with expressions like $A \times D \pm B \times C$ when their values are somewhat smaller in magnitude than Λ even though the individual products are not. The following subprocedure is needed.

Coping with the Determinant $x \times t - y \times z$:
The evaluation of expressions like $x \times t - y \times z = \det\left(\begin{smallmatrix} x & z \\ y & t \end{smallmatrix}\right)$ when x, y, z, t and the determinant are all integers somewhat smaller than Λ in magnitude, even though $x \times t$ and $y \times z$ are both rather bigger, is a subtask that occurs often enough to deserve separate attention. Our approach is inspired by *Gaussian Elimination* except that, instead of seeking a biggest pivot in order to secure numerical stability, it finds the smallest element in the array $\left(\begin{smallmatrix} x & z \\ y & t \end{smallmatrix}\right)$ and reduces some other element to half that size. The reduction process ends either when $x \times t$ and $y \times z$ differ in sign, or when they are both smaller than Λ, in which cases the determinant can be evaluated safely.

```
function Det(x,y,z,t): ... to get det(x z / y t)
    while x × t × y × z ≥ Λ do
        { if | z |>| y | then { s := z;
                                 z := y;
                                 y := s };
          if | x |>| t | then { s := x;
                                 x := t;
                                 t := s };
          if | x |>| z | then { s := x;
                                 x := -z;
                                 z := s;
```

```
                                    s := y;
                                    y := -t;
                                    t := s } ;
            ... now | x |≤| z |≤| y | and | x |≤| t |.
            n := integer nearest y/x;
            y := y - x × n; ... = rem(y, x)
            t := t - z × n; ... = (Det + y × z)/x
            ... now | new y |≤| x/2 | and
            ...     | new t |≤| Det/x | + | z/2 |.
          };
    return Det := x × t - y × z;
    end.
```

Addition and Subtraction:
Like the foregoing functions Product and Quotient, the following procedures act upon two pairs of integers that will pass unchanged through the function Ilt, and the results are pairs that will do likewise provided their magnitudes are somewhat less than Λ.

```
function Sum(A, B, C, D): ... to get (A/B) + (C/D) ilt
    return Sum := Diff(A, B, -C, D);
    end.

function Diff(A, B, C, D): ... to get (A/B) - (C/D) ilt
    G := max{1, gcd(B, D)};    b := B/G;    d := D/G;
    ... Now we seek (A × d - b × C)/(G × b × d) ilt, but first we
    ... must cancel any common factor g hiding in G:
    a := rem(A, G);    c := rem(C, G);    g := gcd(G, a × d - b × c);
    ... Note | a × d - b × c |≤| d × G/2 | + | b × G/2 |< Λ.
    N := (G/g) × b × d; ... the desired denominator.
    ... The numerator will be M = (A × d - b × C)/g ...
    a := rem(a, g);    c := rem(c, g);
    M := (a × d - b × c)/g + Det((A - a)/g, b, (C - c)/g, d);
    ... Note how | a × d - b × c |< Λ as before.
    return Diff := (M, N);
    end.
```

Are they worth the bother?
It seems at first unlikely that a calculation of

$$M/N := A/B \pm C/D = (A \times D \pm B \times C)/(B \times D) \quad \text{ilt}$$

would start with integers A, B, C, D not much smaller than Λ and end with integers M, N no bigger than Λ. But, having programmed the foregoing procedures into various programmable calculators including an HP-97 and an HP-71B, I have seen these unlikely events occur about as often as not. Perhaps this is merely evidence that I have been computing some things the hard way instead of the easy way, rather than evidence that anyone else will use the programs every day.

These programs are the simplest I know that exemplify a property more often found among numerical programs than others; their simple and natural specifications belie complicated and unnatural implementations. It may seem natural to demand that, if the data given a program and the output desired from it can both be represented exactly within the convenient range of a computer's capabilities, the output actually delivered should be correct. But that demand implies that the program will find a path from the data to the output without first transgressing the computer's limitations despite that the path begins and ends only a step or two away from the edge. Such a path need not be obvious.

Programs:

Programs for the HP-67/97 and HP-71B have been appended to these notes. The program for the HP-67/97 requires very little change to run on the HP-41C or HP-15C. Although the HP-71B program is written in a kind of BASIC that looks as if it would run on diverse other machines, the program exploits the HP-71B's conformity to IEEE 854 in two ways. First, its rem operator (called RED on the HP-71B) is built-in and allows the program to handle integer inputs as big as $\Lambda = 100\ 00000\ 00000$. Second, the *Inexact* signal accessible through FLAG(INX, ...) permits the program to try obvious algorithms first and then, only if it encounters roundoff, resort to slower ones. Chained sequences of rational operations can be attempted in confidence because their results will assuredly be correct unless *Inexact* is signaled.

Acknowledgements:

Although prepared in this form for an Introductory Numerical Analysis class, these notes are based upon researches continued over an extended period. The author has used procedures similar to Det in programs that solve linear and quadratic equations, precondition ill-conditioned problems to make them easier to solve accurately, and prepare test data for other programs. That work has been supported at times by grants from the Research Offices of the U. S. Army, Navy and Air Force under contracts numbered respectively DAA629-85-K-0070, N00014-76-C-0013 and AFOSR-84-0158.

HP-67/97 program to perform RATIONAL ARITHMETIC on pairs of integers in Lowest Terms

Usage: The stack holds four integers X, Y, Z, T construed as two rational numbers Y/X and T/Z, both presumed to be *in lowest terms* (ilt). If not, pressing [E] will reduce Y/X to lowest terms while leaving T/Z unchanged. The four rational operations are performed by pressing one of the keys [A], [B], [C], [D] to invoke reliable programs, or [a], [b] [c] [d] to invoke obvious programs. The reliable programs accept integers as large as 1,999,999,999 and deliver exactly correct results up to 8,000,000,000 . Specifically, the programs ...

Add:	Press [A] or [a] to put Y/X:=(T/Z)+(Y/X) ilt, leaving T/Z unchanged.
Subtract:	Press [B] or [b] to put Y/X:=(T/Z)-(Y/X) ilt, leaving T/Z unchanged.
Multiply:	Press [C] or [c] to put Y/X:=(T/Z) x (Y/X) ilt, leaving T/Z unchanged.
Divide:	Press [D] or [d] to put Y/X:=(T/Z) ÷ (Y/X) ilt, leaving T/Z unchanged.
Reduce:	Press [E] to put Y/X:=(Y/X) ilt, leaving T/Z unchanged.
GCD:	Press [e] to put X:= Greatest Common Divisor of X and Y.
REM:	Press [GSB] [8] to put X:=Y-nX and n:= Integer nearest Y/X into reg. 8.

The programs use registers 0 to 8 and I, and labels 2 to 8 too.

```
Program: *LBL A CHS *LBL B GSB 7 X ≷ Y R↑ STO 4 GSB e X=0? EEX STO 5 STO÷0 STO÷4 RCL 1 X ≷ Y
         GSB 8 RCL 4 x STO 6 RCL 3 R↑ GSB 8 RCL 0 STO 7 x RCL 6 - GSB e STO÷5 RCL 1 X ≷ Y
         GSB 8 RCL 4 STOx0 x STO 1 RCL 8 STO 6 RCL 3 R↑ GSB 8 RCL 7 x RCL 1 - X ≷ Y ÷ STO 1
         RCL 5 STOx0 RCL 8 STO 5 *LBL 5 RCL 6 RCL 4 x ENT↑ ENT↑ RCL 5 RCL 7 x x EEX 1 0
         X>Y? GTO 4 RCL 6 ABS RCL 4 ABS X≤Y? GTO 3 LASTX RCL 6 STO 4 X ≷ Y STO 6 *LBL 3 RCL 5
```

```
      ABS RCL 7 ABS X≤Y? GTO 3 LASTX RCL 5 STO 7 X ≷ Y STO 5 *LBL3 RCL 7 ABS RCL 4 ABS
      X≤Y? GTO 3 RCL 6 RCL 7 GSB 8 STO 6 RCL 4 RCL 8 x STO-5 GTO 5 *LBL 3 RCL 5 RCL 4
      GSB 8 STO 5 RCL 7 RCL 8 x STO-6 GTO 5 *LBL 4 LASTX R↑ - STO+1 GTO 6
      *LBL e 3 CHS STO I R↓ X=0? GTO 2 GSB 8 GTO i (jumps back three steps to X=0?)
      *LBL 8 STO 8 X ≷ Y ENT↑ ENT↑ RCL 8 ÷ DSP 0 RND STO 8 R↑ x - RTN *LBL 2 X ≷ Y ABS RTN
      *LBL 7 STO 0 R↓ STO 1 R↓ STO 2 R↓ STO 3 RTN
      *LBL E GSB 7 R↓ GSB e X=0? GTO 6 STO÷0 STO÷1
      *LBL 6 RCL 3 RCL 2 RCL 1 RCL 0 X>0? RTN CHS X ≷ Y CHS X ≷ Y RTN
      *LBL D X ≷ Y *LBL C GSB 7 ST0 4 GSB e X≠0? STO÷0 X≠0? STO÷4 RCL 1 RCL 2 GSB e
      X≠0? STO÷1 RCL 2 X ≷ Y X≠0? ÷ STOx0 RCL 4 STOx1 GTO 6
      *LBL a CHS *LBL b GSB 7 x X ≷ Y R↑ STOx0 x - STO 1 GSB 6 GTO E
      *LBL d X ≷ Y *LBL c GSB 7 STOx1 R↑ STOx0 GSB 6 GTO E
```

```
10 !  Listing of  HP-71B  program to perform  RATIONAL ARITHMETIC
20 !   upon pairs of integers in  Lowest Terms  conveyed as
30 !   "Complex Variables"  to represent  R = M/N  as  (M,N) .
40 !  The  "Complex"  functions herein are ...
50 !    fnA(R,S) = R+S       fnS(R,S) = R-S
60 !    fnM(R,S) = R*S       fnD(R,S) = R/S
70 !    fnI(R) = R in lowest terms (ilt)
80 !  Supporting  Real functions  include ...
90 !      fnDO(R,S) = det(R,S) = Impt(Conj(R)*S)
100 !     fnG(I,J) = Greatest Common Divisor of  I and J .
110 !     RED(I,J) = rem(I,J) = I rem J   as in  IEEE st'd p854
120 ! RUN  to sense  FLAG(INX)  and reset it to  0 ;  if that
130 !   changes then a result has been compromised by roundoff.
140 COMPLEX  R,S, R1,S1, R2,S2, R3,S3, R4,S4, R5,S5, R6
150 ! ***************
160 DEF FNG(I0,J0) ! ... = GCD(I0,J0)
170  IF J0=0 THEN 190
180   o0=J0 @ J0=RED(I0,J0) @ I0=o0 @ IF J0#0 THEN 180
190  FNG=ABS(I0) @ END DEF
200 ! ***************
210 DEF FNI(R6) ! ... = R6 IN LOWEST TERMS
220  FNI=R6/MAX(1,FNG(REPT(R6),IMPT(R6)))*SGN(CLASS(IMPT(R6)))
230  END DEF
240 ! ***************
250 DEF FNDO(R5,S5) ! ... = det(R5,S5) = Impt(Conj(R5)*S5)
260  o1=REPT(R5) @ o2=IMPT(R5) @ o3=REPT(S5) @ o4=IMPT(S5)
270  o0=FLAG(INX,0) @ o5=SGN(o1*o4)*SGN(o2*o3) @ o0=FLAG(INX,o0)
280   IF o0=0 OR o5#1 THEN 350
290    IF ABS(o3)>ABS(o2) THEN o5=o3 @ o3=o2 @ o2=o5
300    IF ABS(o1)>ABS(o4) THEN o5=o1 @ o1=o4 @ o4=o5
310    IF ABS(o1)<=ABS(o3) THEN 330
320     o5=o1 @ o1=-o3 @ o3=o5 @ o5=o2 @ o2=-o4 @ o4=o5
330    o5=RED(o2,o1) @ o0=(o2-o5)/o1 @ o2=o5 @ o4=o4-o3*o0
340    GOTO 270
350  FNDO=o1*o4-o2*o3 @ END DEF
360 ! ***************
370 DEF FNM(R4,S4) ! ... = R4*S4 in lowest terms
380  o1=REPT(R4) @ o2=IMPT(R4) @ o3=REPT(S4) @ o4=IMPT(S4)
390  o5=MAX(1,FNG(o1,o4)) @ o0=MAX(1,FNG(o2,o3))
400  FNM=((o1/o5)*(o3/o0), (o2/o0)*(o4/o5)) @ END DEF
410 ! ***************
420 DEF FND(R3,S3)=FNM(R3,(IMPT(S3),REPT(S3))) ! ... = R3/S3 ilt
430 ! ***************
440 DEF FNS(R2,S2) ! ... = R2-S2 in lowest terms
```

```
450  o1=REPT(R2) @ o2=IMPT(R2) @ o3=REPT(S2) @ o4=IMPT(S2)
460  o0=FLAG(INX,0) @ o5=o1*o4-o2*o3 @ o6=o2*o4 @ o0=FLAG(INX,o0)
470  IF o0=0 THEN FNS=FNI((o5,o6)) @ GOTO 530
480   o9=MAX(1,FNG(o2,o4)) @ o2=o2/o9 @ o4=o4/o9
490   o6=RED(o1,o9) @ o7=RED(o3,o9) @ o5=FNG(o9,o6*o4-o2*o7)
500   o9=(o9/o5)*o2*o4 @ o6=RED(o6,o5) @ o7=RED(o7,o5)
510   o8=(o6*o4-o2*o7)/o5 @ o1=(o1-o6)/o5 @ o3=(o3-o7)/o5
520   FNS=(FND0((o1,o2),(o3,o4))+o8, o9)
530  END DEF
540 ! ***************
550 DEF FNA(R1,S1)=FNS(R1,(-REPT(S1),IMPT(S1))) ! ... = R1+S1 ilt
560 ! ***************
570 IF FLAG(INX,0)=0 THEN DISP "Exact" ELSE DISP "Inexact"
```