FLOATING-POINT ALGORITHMS

as used at the Dept. for Automation in Engineering, Computer and Automation Institute, Hung. Acad. of Sci.

Third printing, revised

Budapest 1985

First printing: 1979
Second printing, enlarged: 1982

Készült
a Magyar Tudományos Akadémia Számitástechnikai
és Automatizálási Kutató Intézetében

Felelős kiadó

DR. VÁMOS TIBOR

ISBN 963 311 186 2

Hozott anyagról sokszorosítva

8515409 MTA Sokszorosító, Budapest. F. v.: dr Héczey Lászlóné

CONTENTS

0. PREFACE

This is to accompany specifications of various pieces of soft- and hardware (on Intel 8080, Motorola 6800, PDP 11/40 in TPA 70 FPU, GD'80, etc.), making their floating-point features more understandable. Allusions to implementations the reader is unacquainted with may be neglected and skipped. Only crucial points are commented upon, details felt trivial are bypassed. [Unnecessary remarks are square-bracketed throughout.]

All these implementations are based on a fairly coherent set of algorithms. Algorithm design steers a middle-course between a long code and a slow execution; accuracy has not been drawn into compromises. Especially, number representations and operations are shaped to carry as much precision as theoretically possible, rounding is done whenever it should be, and measures are taken to reduce error accumulation while computing standard functions.

Algorithms will be described first for implementations based on hardware microcoding; peculiarities with pure soft-coding (they are few in number) are set forth afterwards. [It is not claimed that a logical distinction can be established between coding and microcoding; differences between soft and hard are vanishing. Nevertheless, there *are* differences.]

1. NUMBERS

Number representation is as follows. (Base 16 numerals are prefixed by a colon.)

A Number takes normally 32 bits (48 and 64 bit representations are used in some cases, but in what follows we will confine us to the 32 bit case, differences being obvious). A Number has a Sign part, an Exponent part, and a Fractional part. S is one bit, E is eight bits, F occupies the rest.

A Value is represented this way. A Value of zero is represented by a Number with E = 0. More accurately: a zero *produced* by the algorithms is always all-zeros; a Number *accepted* by the algorithms is considered zero if only E = 0 [irrespective of S and F; so there are 2^{24} patterns that can carry extraneous information but are taken as 0 within floating-point; there is a convention to consider "minus zero", that is, :80000000, as "floating nil"]. A non-zero Value can be written, and in a unique manner, as $\pm\ 2^e \cdot f$ where f is less than 1 and not less than 1/2, and e is an integer (in fact e is $[\log_2|v|]+1$ and f is $2^{\log_2|v|-[\log_2|v|]-1}$). A non-zero Value is said to be (exactly) *representable* if $e \in [-127,+127]$ and $f \cdot 2^{24}$ is an integer ($\in [2^{23},2^{24}-1]$). Now a representable Value is represented by a Number with $S = \{{0 \text{ for } +}\atop{1 \text{ for } -}\}$, $E = e+128$, $F = f\cdot 2^{24}-2^{23}$, all these meaning binary integers. [So F fits into 23 bits. Put it in another way: Regard f as a binary fraction, binary point on the left, then it is always "normalized", that is to say, its leftmost bit is always a 1, thus it is redundant and need not be stored. This is called the "hidden bit" notation. As to E, there is no special refinement about it; any binary representation not using zero (or, leaving out one value) would do; this one is one of them, and fairly convenient. It is

called "base 2, excess 128" notation.] A Value that does not satisfy the condition on f gets *rounded* to a Value that does; a Value not satisfying the condition on e will be set, by convention, to "machine infinity" ($e = 127$, $f = 1-2^{-24}$) for $e > 127$, and to zero for $e < -127$.

An *unpacked* Number is the triad of the Number's separated component parts: its sign set apart, its exponent as a byte, and its fractional part in three bytes with leftmost bit set (even for a zero). *Those* will be called S, E, and F, resp.

[This representation had been invented by Zuse (as early as 1944). Its most widespread computer application is perhaps that in the PDP 11/45 floating-point hardware. The argument for preferring it is its added accuracy. Compared, e.g., with a base 16 representation, as used with IBM, where a fractional part may begin with three zeros (which means a loss in accuracy of three binary orders of magnitude in the worst case, and arithmetic can hardly be considered more accurate than its worst case) and where, in addition, no such thing as a "hidden bit" is possible, we have a handicap of four binary places if the same storage size is used for fractional parts. If rounding arithmetic is acknowledged to furnish "half-a-bit" additional accuracy, which can be argued on good reasons, then rounding Zuse can be said to have about 22.6 times more "fractional precision" than non-rounding base 16 representation. This can but partly be compensated by increasing the size of the fractional part at the expense of the exponent. (IBM has in fact a 7 bit exponent and a 24 bit fraction. So, e.g., the least representable value greater than 1, which is $1+2^{-23}$ with PDP, is $1+2^{-20}$ on IBM, and, in addition, because of non-rounding, any result but slightly less than $1+2^{-20}$ will truncate to 1. Surely, even base

16 exponent representation cannot be reduced to less than 6 bits without making the exponent range uncomfortably small. With 6 bits - which would roughly equalize the IBM exponent range to that of PDP - write 2^{-21} in place of 2^{-20} above, and remind that a 25 bit fractional part is not very handy on a byte machine.) - As to balancing fraction against exponent in Zuse, an 8 bit binary exponent admits of a ratio of more than 10^{76} between the largest and the smallest-non-zero absolute value (which, leaving apart von Neumann's argumentum ad hominem that he who knows his problem knows his orders of magnitude, will be likely to comprise quite automatically all possible spans in any practical problem), while, on the other hand, the corresponding 24 bit fractional precision is so to say on a critical border, guaranteeing slightly more than seven decimal digits, thus arithmetic may remain confined to 32 bit numbers for most present-day industrial, engineering, physical, etc. applications (where input data are seldom known and output data seldom needed to more than five decimal digits), even allowing for cumulative errors (which, if independent, and unaided by the programmer, eat up two decimal places for at about ten thousand operations - a frivolous guess as to the number of times something happens to anything in a program). As it is fairly common practice in engineering etc. computations to escape, at least partly, from ordinary 32 bit arithmetic into "double precision" (nobody's daydream, and a nightmare for anybody in need of the small-and-speedy), this can pass as an argument in favour of Zuse.]

2. BASIC OPERATIONS

Operations are defined in a way that they yield a (possibly sensible) result for any operands [as usual with integer arithmetic] instead of making some cases "illegal" or "exception" [as under most floating-point rêgimes]; in cases commonly considered errors the programmer has the choice to "detect" them or to go on with the outcome if he likes it. So overflow with the four species and division by zero produce signed "machine infinity" *and* set the Overflow bit (which can be tested both from within the program and from "trace" routines, described elsewhere). Underflow produces zero, without setting anything but the zero bit. [This, however, could be reconsidered; setting Overflow would be unambiguous and sometimes of interest. - A point worth of consideration is to have (both for floating and non-floating) a "sticky" Overflow bit: remaining set until tested.] Square root of negative is defined negative square root of absolute.

Zero and Minus bits are set in the unsurprising way.
- Where there is a separate Carry bit (as distinguished from Overflow), it gets cleared by floating-point operations.

Working is described in terms of three "registers": R, P, and Q, each taking four bytes, and capable of being used more or less slice-wise: left byte separately, the three remaining bytes separately, and all four as a whole. If the left byte is referred to, the register name will be subscripted with E; if the three right bytes together, with F. R is often physically identical with the destination register, if any; P is a work-register; Q is what is called Q in bit-slice chips (e.g. Am 2901). [Many of the algorithmic idiosyncrasies are explained by the need of

forcing 25 bit precision upon fraction computations using such register sizes, and this without much shifting around and without using more than one work-register. - So-called "single-instruction cycles" are used whenever possible.]

Two's-complement arithmetic is used.

Logical *true* is represented by 1, *false* by 0.

X is defined as a subsidiary result of addition and subtraction or comparison of operands taken as unsigned binary integers: For subtraction and comparison, X is 1 if the second operand has been greater than the first, and 0 otherwise; for addition, compare the one's complement (logical NOT) of the first operand with the second operand. [To bring it under a common formula, imagine both operands appended on the left with a zero bit, then X is the corresponding bit of the sum or difference, resp. - With addition, this X is what is commonly produced as "carry". With subtraction, customs diverge; e.g. Intel 8080's carry is our X, Am 2901's is its complement.]

Y is the leftmost bit.

2.1. SIMPLE SINGLE OPERATIONS

Operand in Q; result in Q.

Negate, Absolute, Sign, Clear, Test are evident. Some implementations work on unpacked, others on non-unpacked Numbers. Some of them forget, for shortness through uniformity, that Clear needs no getting, and Test no setting.

Integer-Part, Fractional-Part yield floating-point results; they are defined [not in the mathematical tradition but] as 0 for op = 0, otherwise $\frac{op}{|op|}\cdot[|op|]$ and $\frac{op}{|op|}\cdot(|op|-[|op|])$, resp.; their coding requires no peculiar elucidation.

2.2. THE FOUR SPECIES

First operand in R, second operand in Q; result in R.

At start, operands get unpacked; W, the antivalence (exclusive-OR) of the two S's, is produced and stored away; E and F of the first operand go to R_E and R_F, similarly the second to Q.

At end, operations gather at four points: CLEAR attracts all cases of a zero result, sets the result zero, the Zero bit one, and the Minus bit zero. All other cases clear the Zero bit. OFLOW attracts overflow, sets the result to signed "machine infinity", the Overflow bit to one, and permits the Minus bit to take its value. [Zero and, of course, overflow are special enough, both in representation and action, to deserve a separate handling; thus the remaining "normal" cases undergo a simpler and more uniform treatment. There are in fact so many places to discover a zero result that in microcodes, which require no separate instruction for "return"-ing, the address of CLEAR is put on the stack top.] In the remaining cases, R will contain the result's E and F in one of two possible states: either according to the form called unpacked, then it goes to SET, or in a form that differs from unpacked in that the fractional part is shifted one place to the left (the "hidden bit" is shifted out), then it goes to ROUND. At SET, a left shift is made over R_F coupled with Q_F (which contains the "continuation" of the result), to go on at ROUND. ROUND rounds, by adding 1 at the rightmost position of R. [Note that the exponent need not be handled separately; if the addition is performed on R_E and R_F taken together as a whole, that is, on R, then what happens is just rounding.] Again, we have escapes to OFLOW [since rounding may cause overflow] and to CLEAR [since it is convenient to let flow this way some cases which may be 0].

All what remains to do is a right shift moving in the value of the Minus bit (containing already the sign of the result).

ADDITION

For a zero second operand, the first operand (contained in R) is the result. Otherwise R is going to take, or has already taken, an operand whose absolute value is not less than that of the other operand; its sign will be the result's sign; P takes the other operand. Along with this, the cases "difference of equals" (result is zero) and "not-greater operand is zero" (result is in R) are singled out. Similarly, if R_E exceeds P_E by more than 25, we are ready (result in R). [Reason for making these cases special is partly the devious representation of zero, partly the resulting ease at normalization (having always a bit to twang), partly average time saving by avoiding up to 255 shifts whilst P has no influence on the result.] Now shift P_F coupled with the previously cleared Q_F to the right R_E-P_E places (may be 0). [This is a single-instruction cycle.] Then branch on W to "add" or "subtract". [An alternative solution could have been to bring back "subtract" to "add" by negating; this has been rejected (not for its arithmetical clumsiness but) because "add" and "subtract" take distinct courses afterwards.] (A) "add": Add. [This can be done by clearing P_E and adding over the whole of R and P provided the X on the "F part" of the addition is separately available; the resulting E will be incremented if it has to.] Then if X on E is 1 go to OFLOW; if X on F is 1 go to ROUND; else go to SET. [Obviously so; remember that Q will be used, and properly, in the last case.] (B) "subtract": Negate (by subtracting from zero) the contents of Q_F; subtract P_F and X on the Q-operation from R_F; normalize R_F coupled with Q_F [again

a single-instruction cycle] and deduct the number of steps from R_E. ["Overshooting" cycles are used here (and elsewhere) for pipelined hardware, where PC-incrementing is delayed.] If X on E is 1 or R_E is 0 then there is underflow (that is, CLEAR), else go to ROUND.

SUBTRACTION

is reduced to addition by inverting the S bit of the second operand.

MULTIPLICATION

The result's sign is W, except for a zero result (that goes through CLEAR). If either of the E's is zero the result is zero; otherwise the sum of the E's minus 129 is the "tentative exponent" of the result [where a 0 does not yet mean underflow]. [Complications in coding are due to being confined to byte-arithmetic.] The F of the first operand goes to P_F, that of the second remains in Q_F; R_F gets cleared. Now the following is done 24 times: If the rightmost bit in Q_F is set add P_F to R_F, if it is not set do nothing (add 0 to R_F); then in both cases shift right R_F coupled with Q_F, moving in X on the left. [Hardware has been designed in a way that all this become a single-instruction cycle, doing different things under different conditions. - Another approach could have been to multiply the other way round, with the eventuality of breaking off earlier in case of trailing zeros, a slight, but not cheap, improvement. - As to X, it makes up for a leftmost "work-bit" to hold a "carry".] Now add the last X to the "tentative exponent" to produce the exponent (it may OFLOW, and it may remain 0 in which case it will find its way to CLEAR after ROUND if

it keeps underflown); for X = 0 shift left one place; then go to SET (or, equivalently, shift left and go to ROUND).

DIVISION

Sign is the same as with multiplication. If the second E is zero go to OFLOW; if the first E is zero go to CLEAR. The "tentative exponent" is first-E minus second-E plus 128. [A great deal of sign and exponent handling in multiplication and division shares a common piece of code.] R_F is to hold the F of the first operand, P_F that of the second operand; Q_F gets cleared. Compare R_F with P_F and add the complement of X to the "tentative exponent" to produce the exponent; if it overflows go to OFLOW, if it remains 0 go to CLEAR (being underflown) [rounding cannot cure underflow (nor cause overflow) with division]; for X = 0 shift right R_F coupled with Q_F, moving in 0 on the left. Let T denote the equivalence (exclusive-NOR) of the current result's X and the previous result's Y, both taken on the "F part". Execute something that causes Y = 0, then something that causes Y = X = 0 [of course they are free to do other good work meanwhile]. Now the following is done 26 times: If T is 1 subtract P_F from, otherwise add P_F to, R_F; then shift left R_F coupled with Q_F, moving in the (nascent new) value of T on the right. [Again, hardware has been designed to do all this in a single-instruction cycle. - We have one step more than logically necessary; avoiding this costs even more. - What really happens is "non-restoring division" (a well-known technique not described here) with some kinks. We ought to find a place for the sign. As we have no bit to waste, so make sure the following assertion holds true: The complement of the sign in normal non-restoring division is identical, at each step, with our T resulting from the same sequence of operations (additions or

substractions, as they come) performed on the operands devoid of sign position, if only pre-Y, X, and Y have been duly initialized. So this holds through, and the same Q is built. - This T is, after all, virtual space, like X with multiplication. It *has* to be stored, though; but it is *one* bit instead of many, and can be placed *outside* the bit-slices - this being the point.] Now move Q_F to R_F, and go to ROUND.

On bit-slice processors broader than needed no X-Y play and no T is implemented, and the sign bit is used in the obvious way.

COMPARISON

is not reduced to subtraction but coded separately.

[The Overflow bit could be used to indicate if the first operand is smaller than the second in absolute value; this might prove useful. - Detecting relative error, if smaller or not than a writable "epsilon", has been dismissed as unusual but can be advocated as desirable.]

MULTIPLY-AND-TAKE-FRACTIONAL-PART

needs no explanation. [It may add to fractional precision.]

DOUBLE and HALVE

are single-operand operations - operand in Q; result in Q. They multiply/divide by 2, manipulating the exponent; overflow/underflow yield signed "machine infinity" and zero, resp.

The last three operations are not present in all implementations.

2.3. SQUARE ROOT

Square root *is* a single-operand operation: Operand in Q; result in Q.

Zero remains zero. Else, S is stored (square root of negative is negative); (E+129) ÷ 2 ("÷" is integer-division) will be the result's E - store it [we will use the E-parts, too, for fractional computation]. F goes left-shifted to R_F, R_E is set to 1 [this is the "hidden bit"]; P takes :01000000. If the original E was even, shift left R (or P). Set Q to the value of P. Now the following is done 25 times (or until P is found zero): Compare R with Q; for X = 0 subtract Q from R and add P to Q, otherwise subtract P from Q; then shift left R, shift right P, and add P to Q. [Available bit-slice chips cannot be induced to execute *that* as a single-instruction cycle.] Then shift right Q, put in the exponent, and we are in the pre-ROUND state - with Q instead of R. [In versions where single-operands have not necessarily a destination register, the contents of R are saved and restored.]

[To verify the method, recall the paired-digit hand-computing of the square root. - First we were satisfied that Gaussian iteration, i.e., Newton-Raphson, is the optimal method of taking the square root, which it certainly *is* - *if* the primitives are the four species. As we have deeper and cheaper primitives, the above is more optimal, in fact very optimal. (Remarkably, Hewlett & Packard use an analogous algorithm over a base 10 representation.) - The availability of a really "basic" square root influenced, moreover, the design of "standard function" algorithms, encouraging us to uniformize them the way we did; see exp.]

3. STANDARD FUNCTIONS

[Which functions are standard is a matter of convention. A small set has been chosen; some others can be reduced to those. Formally, standard functions are single-operand operations, called by programs the same way. Nevertheless, the distinction and separate treatment has been retained, for our standard functions have a lot of common features.]

Functions present in all implementations are: sin, cos, arctan, log, exp. Others are added in some cases. - "Impossible" arguments are disposed of in the following way: log of negative equals that of absolute, log of zero is minus "machine infinity"; exp gives "machine infinity" for too big, and its reciprocal for too small.

Modes are: radian and degree for trigonometric functions, natural, binary, and decimal for logarithm and exponential. These are dealt with in an obvious way at start or at end, as appropriate; computation runs in radian and in natural.

Functions come in two versions, Epic and Dramatic. Which one is implemented depends on circumstances (cf. Sect. 3.3.).

3.1. EPIC VERSION

Standard function operands are manipulated to a point where the task consists in computing a Hornerized starting section of a Taylor series. This is done by a routine common to all functions. Its input information is: a - alternate or not; n - number of terms; c - an array of coefficients; and x - the variable. What it does is this: $s := x^2$; $y := 1$; for i := 1 step 1 until n do $y := (-1)^a \cdot y \cdot c[i] \cdot s + 1$; the result being in y. It uses a special number representation (so the |x| value must be converted at start, the y reconverted at end; coefficients are given in this form): four-byte binary fraction with binary point after the leftmost bit. [Thus values between 0 and $2-2^{-31}$ can be represented. - Reasons for having this representation are: (a) Accuracy. Virtually no rounding errors accumulate in a series of operations. (b) Speed. There is no intermediate packing (but at start and at end), no exponent handling, no checking (since occurring data are such as to preclude malfunction); addition-subtraction reduce to manipulating the leftmost bit. Therefore code, too, may be short.] - A slight improvement in computing the "direct" functions (sin, exp) is to use half the x and four times the coefficients in the Horner scheme, halving of x being made within the internal four-byte representation. (This is missing from earlier implementations; differences in coding are obvious.) - The operation is always followed by computing y·x. This is sometimes coded as part of the routine. - Shifting may precede squaring. For $|x| < 2^{-15}$ skip all this. - The routine is accessible from outside [maybe someone likes it].

[In what follows, everything depends on whether ranges have been properly established, truncation errors properly estimated, etc. So it is; and sin and arctan happen to require

n = 5, exp and log, n = 4. (Thus n is a+4, a is n mod 2.) Calculations, being rather voluminous, have been omitted, but can be easily re-produced by the interested; some hints are given. Comparisons with Chebyshev and other methods are also left to those concerned with. - In some cases, some bias in a Horner coefficient may improve upon precision by adjusting the results in a way that makes a higher-power term superfluous or a specific critical value more exact. This device has no application, however, with *these* functions in *this* arithmetic; it has occasionally been used in the 64 bit arithmetic.]

Reals mean their floating representation.

SIN

Remember the sign, s; take absolute value, call it x. Take it mod 2π; if x is $\geq \pi$, subtract π from it and invert s; then if x is $\geq \frac{\pi}{2}$ replace it by $\pi - x$. Now HORNER for half-x: a = 1; n = 5; c = :04A7904A (4/110), :071C71C7 (4/72), :0C30C30C (4/42), :1999999A (4/20), :55555555 (4/6). Then multiply by x and sign it s; this is the result.

Weak points are: operands of great absolute value, with a blurry remainder by mod. [This is necessary, and inherent in limited-precision arithmetic; the loss in significant digits is roughly proportional to the logarithm of the operand's absolute value.]

COS

is played over to sin, by adding $\frac{\pi}{2}$ to the operand.

(Some codings compute cos directly and sin through cos.

This has the advantage that a Horner setup with n = 4 only is needed, and the disadvantage that the code is less uniform, and that sin, especially near 0, is less accurate. [Obviously. Cf. the remarks on a related problem raised by the computation of exp, below.])

ARCTAN

Remember the sign, s; take absolute value; call it x. Set t = 0; if x > 1 replace it by 1/x and set $t = \frac{\pi}{2}$. Set u = 0; if $x > 2 - \sqrt{3}$ replace it by $\frac{x - 1/\sqrt{3}}{x/\sqrt{3} + 1}$ and set $u = \frac{\pi}{6}$. Now HORNER for x: a = 1; n = 5; c = :68BA2E8C (9/11), :638E38E4 (7/9), :5B6DB6DB (5/7), :4CCCCCCD (3/5), :2AAAAAAB (1/3). Then compute $|x \cdot y + u - t|$ and sign it s; this is the result.

LOG

Take care of 0; take absolute value; call it x. Take E-128 of x; float it; subtract 1/2; multiply by log 2; call it t. Set E of x to 128; set x to $\frac{x - 1/\sqrt{2}}{x + 1/\sqrt{2}}$. Now HORNER for x: a = 0; n = 4; c = :638E38E4 (7/9), :5B6DB6DB (5/7), :4CCCCCCD (3/5), :2AAAAAAB (1/3). Then multiply by 2x and add t; this is the result.

[The underlying formula is

$$\log \xi = -\log\sqrt{2} + 2\sum_{h=0}^{\infty} \frac{1}{2h+1} \left(\frac{\xi - \frac{1}{\sqrt{2}}}{\xi + \frac{1}{\sqrt{2}}}\right)^{2h+1}$$

(which can be derived from the series for $\log(1 \pm \xi)$). So we compute log essentially by an "arctan series", which, unlike the series for $\log(1 + \xi)$, converges very rapidly, and fits, moreover, our Horner pattern.]

EXP

Remember the sign, s; take absolute; divide by log 2; call it x. Set t = integer part of x; set x = fractional part of x multiplied by log 2. Now HORNER for half-x: a = 0; n = 4; c = :071C71C7 (4/72), :0C30C30C (4/42), :1999999A (4/20), :55555555 (4/6). Then multiply y by x [we computed in fact sinh, now we have to find back]; compute $y + \sqrt{(y^2+1)}$; increment its E by t "fixed" (or set it t + 129), overflow meaning "machine infinity" (in some codings, this turns out previously); take its reciprocal for a negative s [which is not absolutely nice in case of an operand of very large negative value since the smallest positive representable value is smaller than the reciprocal of the largest; but so it stands]; this is the result.

[Computing sinh rather than exp has the advantage that it requires half the number of terms and it conforms our Horner scheme. Consequential square-rooting, being fast, does but partly play away the gain. - Had we really "reals" instead of a floating-point, we could have used cosh instead of sinh, and save a multiplication. Near zero, however, cosh is rather flat as compared with exp, so recomputing this from that amounts to losing information (or, which is the reverse of the same thing, with cosh we ought to go through $y^2 - 1$, a difference of near-equals); the outcome would be something like a "step function": constant for a while, then high-jumping to the next level.]

*

Some constants as represented in "Zuse":

$1/\sqrt{2}$	≈	:4035O4F3
$1/\sqrt{3}$	≈	:4013CD3A
$2-\sqrt{3}$	≈	:3F893OA3
log 2	≈	:40317218
log 10	≈	:41135D8E
π	≈	:41490FDB
π/6	≈	:40060A92

3.2. DRAMATIC VERSION

This method yields a *very* short code at the cost of a somewhat larger number of instructions to be executed. [As the method is not immediately self-evident, we are going to give a more detailed treatment.]

NOTIONS, NOTATIONS

Functions to be computed are sin, cos, sinh, cosh, exp, arctan, artanh, log. - The first five will be called "direct" functions, the rest, "inverse" functions; sin, cos and arctan will be called "trg" functions, the rest, "hyp" functions. - All direct functions get computed by essentially the same algorithm, all inverse functions by another one. sin and cos are produced simultaneously; similarly sinh and cosh. [It takes practically no longer to have them both than one alone.] exp and log get reduced to the restly functions by the relations $\exp x = \cosh x + \sinh x$ and $\log x = -2 \operatorname{artanh} \frac{1-x}{1+x}$. [This apparent byway will turn out a shortcut.] So we are going to have two setups: one for sin, cos, sinh, cosh, and one for arctan and artanh; these, too, may have shared parts.

We define the "standard domain" of the argument as

$[0, \frac{\pi}{4}]$	for sin and cos;
$[0, 1]$	for arctan;
$[0, \log 2)$	for sinh, cosh and exp;
$[\frac{1}{2}, 1)$	for log;
$(0, \frac{1}{3}]$	for artanh.

We assume that arguments have been "reduced" to their standard domains. (To do this, we have the sign rules, we have modulo, negating, mirroring, interchanging for sin and cos, and we have $\arctan x = \frac{\pi}{2} - \arctan \frac{1}{x}$, $e^x = 2^{[\frac{x}{\log 2}]}\, e^{\log 2(\frac{x}{\log 2} - [\frac{x}{\log 2}])}$ and $\log x = ([\log_2 x]+1)\log 2 + \log(2^{-[\log_2 x]-1} x) = e \log 2 + \log f$ (for e and f see Sect. 1); artanh, if used for log, will fall in the domain. [These reductions are analogous to, but partly different from, those performed in the previous version. - Wider or narrower domains are possible; they don't affect the general formulae below, only the number sizes to be represented and the number of computation steps.])

Throughout the general discussion we assume, moreover, that the argument is *not* zero; it will turn out, in retrospect, that a zero argument, too, whenever permitted, may run through the algorithms. [It can of course be handled separately if preferred.]

We use the following notations:

x - the argument.

$\bar{c} := -[\log_2 x]-1$; this is the "negexponent" of the argument, that is, the negative of our e; obviously $2^{-(\bar{c}+1)} \le x < 2^{-\bar{c}}$. [Remember $x \neq 0$.]

$\bar{y} := 2^{\bar{c}} x$; so $\bar{y} \in [\frac{1}{2}, 1)$; it is our f, the full "fractional part" of x (with "hidden bit" shown).

Set $c := \bar{c}+1$ for trg, and $c := \bar{c}$ for hyp; set $y := 2^c x$.

All the algorithms accept c and y as input. They produce $2^c \sin x$, $\cos x$, $2^c \sinh x$, $\cosh x$, $2^c \arctan x$, $2^c \operatorname{artanh} x$, resp.

If the arguments have been within their standard domains, all the results will be $\in [\frac{1}{2}, 2)$. Numbers in $[\frac{1}{2}, 1)$ are considered "normalized"; to normalize results, (at most) one halving (right-shift) may be needed.

h is the number of significant bits required. (Normally h = 25, to get a well-rounded 24-bit fractional part.)

m is the "width" of the arithmetic used by the algorithms, that is, the number of bits in the computer "word". Numbers are represented without sign, with binary point after the leftmost bit, so numbers in $[0, 2-2^{-(m-1)}]$ can be represented. Elementary operations are understood without rounding; rounding will be done only at the very end, during the normalization phase. (Normally m = 32.)

n is the "precision prescription" for the computation, requiring a computational error less than $\frac{1}{2^n}$.

$n \geq h$ clearly guarantees h significant bits in the results; for results that are going to be right-shifted, $n \geq h-1$ is sufficient. - m is to be chosen accordingly; it must be obviously $\geq n+1$, and even more, because of cumulative errors. Algorithm analysis shows that error propagation is well behaved; $m \geq h+\log_2 h-1$ would suffice. [So this does not force m = 32 for h = 25; 32 is simply the next practical size. - On the other hand, saving one bit out of m amounts to an overall time saving of nearly $\frac{1}{m}$; hence it may be well worth economizing on m even within a larger physical word size where feasible. (Similar remarks apply of course to the "Epic Version".) - Maybe an even smaller m would do, as experiments indicate; it seems hard to pinpoint the best-possible m; for small h, it might be tested out.]

k is the "depth" of algorithms, that is, the number of steps they have to take. It is nonnegative by definition, and will be chosen positive (at least 1) for technical convenience. Moreover, accuracy imposes lower bounds on k. [Any k greater than these would do; but running time is almost proportional to k, so it is wise to have it as low as possible.] The standard choice for k is: the least positive integer $> \frac{h-5}{4} - \bar{c}$ for direct, $> \frac{h-3}{4} - \bar{c}$ for inverse functions. Some codings have k independent of $\bar{c}$; requirements are: $> \frac{h-6}{4}$ for sin-cos, $> \frac{h-7}{4}$ for sinh-cosh, $> \frac{h-3}{4}$ for arctan, $> \frac{h-9}{4}$ for artanh, which are better for small $\bar{c}$, worse for large $\bar{c}$. [We might have used the smaller one, according to case, but this has not been done.]

THE COMPUTATION

Now the algorithms.

S, T, R, Q are m-bit words, d is a bit, "true" is 1, "false" is 0.

Asterisked signs denote themselves for trg, their opposites for hyp.

D (sin-cos|sinh-cosh)

At start, S is $y = 2^c x$, k is "big enough". At end, S (and R) is $2^c \sin x | 2^c \sinh x$, T is $\cos x | \cosh x$.

```
T ← 1 *- 2^-1 (2^-c-k S)^2; S ← S(1 *- 1/3 2^-1 (2^-c-k S)^2);
for k ← k-1 while k ≥ 0 do
R ← ST; T ← 1 *- 2^-1 (2^-c-k S)^2; S ← R od;
```

I (arctan|artanh)

At start, S is $y = 2^c x$, k is "big enough". At end, S is $2^c \arctan x \,|\, 2^c \operatorname{artanh} x$.

```
for j ← 0,1,..., k-1 do
    S ← 2^-1 S;  T ← 2^-1;  R ← *+2^(-2c-2j-1) S;
    m times do
      R ← 2^-1 R;  T ← T+R;  d ← T≤S;  Q ← 2Q+d;
      S ← 2(S-dT);  T ← T-(-1)^d R od;
    S ← Q od;
S ← S(1-*1/3(2^-c S)^2);
```

If $c > \frac{m-2}{2}$ then it is irrelevant how big it is, but it can waste a lot of time in shifting. So supernumerary shifts are discarded. (This is not shown above.)

In practice, the code is even simpler. Initialization of S in "D", terminalization of S in "I" and T-setting in "D" may use a common piece of code; multiplications by powers of two are, of course, shifts; adding 1 is done by inverting the leftmost bit; the ugly-looking d is readily coded as a condition.

On a well-embedded microcode device, the different working modes of the algorithms can be organized without any branches [by means of the *same* conditionally repeated and self-modifying instructions that have been spoken of in connection with the four species]. - Details vary with equipment idiosyncrasies. (So c is mostly represented :80-c; most implementations produce the left-rotated complement of Q instead of Q; microcodes activate a half-product, not a product, routine; etc.) It should not be very hard to find out the correspondences.

THE JUSTIFICATION

Now we proceed to show that the algorithms really do what they are expected to.

sin-cos

Define

$$s_j := 2^{c+j} \sin \frac{x}{2^j} = 2^{c+j} \sin \frac{y}{2^{c+j}} \; ; \; t_j := \cos \frac{x}{2^j} = \cos \frac{y}{2^{c+j}} .$$

We have

$$s_j = s_{j+1} t_{j+1} \; ; \; t_j = 1 - 2^{-1} (2^{-c-j} s_{j+1})^2 \; ,$$

being a simple application of the well-known relations $\sin x = 2 \sin \frac{x}{2} \cos \frac{x}{2}$ and $\cos x = 1 - 2 \sin^2 \frac{x}{2}$.

What we need is s_o, t_o .

What the algorithm does, for any k, is this.

$$S_k \leftarrow y(1 - \tfrac{1}{3} 2^{-1} (2^{-c-k} y)^2); \; T_k \leftarrow 1 - 2^{-1} (2^{-c-k} y)^2;$$

for $j \leftarrow k-1, k-2, \ldots, 0$ do

$$\{S_j \leftarrow S_{j+1} T_{j+1} \; ; \; T_j \leftarrow 1 - 2^{-1} (2^{-c-j} S_{j+1})^2\}$$

That is, we approximate s_k and t_k by the first two terms of their resp. Taylor series, and compute approximations to s_o and t_o by means of the functional relations of s_j, t_j.

To estimate the approximations, define

$$\sigma_j := s_j - S_j \; ; \; \tau_j := t_j - T_j \; .$$

Then, for $j \le k-1$,

$$\sigma_j = \sigma_{j+1}t_{j+1} + \tau_{j+1}s_{j+1} - \sigma_{j+1}\tau_{j+1};$$

$$\tau_j = -2^{-2c-2j-1}(2\sigma_{j+1}s_{j+1} - \sigma\tfrac{2}{j+1}).$$

From the Taylor series, $0 \le \sigma_k \le 2^{c+k}\frac{1}{5!}(\frac{y}{2^{c+k}})^5 = \frac{16}{15}2^{-4c-4k-7}y^5$;

$0 \le \tau_k \le \frac{1}{4!}(\frac{y}{2^{c+k}})^4 = \frac{4}{3}2^{-4c-4k-5}y^4$; $0 \le \sigma_{k-1} \le 2^{c+k}(\frac{1}{5!}+\frac{1}{4!})(\frac{y}{2^{c+k}})^5$

$= \frac{8}{5}2^{-4c-4k-5}y^5$. (We used that the series alternate.)

For $j \le k-2$,

$$\sigma_j = \cos\frac{y}{2^{c+j+1}}\sigma_{j+1}$$

$$-(2\sin(\frac{y}{2^{c+j+1}} - 2^{-c-j}\sigma_{j+1})(\sin\frac{y}{2^{c+j+2}} - 2^{-c-j-3}\sigma_{j+2})\sigma_{j+2}.$$

(Put the expression for τ_{j+1} into that for σ_j, and write out s and t.)

Let us show that, for all j, $|\sigma_j| \le \sigma_{k-1}$. This being true for $j = k, k-1$, we may assume it true in the right-hand side. For $j \le k-1$, we will need a stronger lower bound, say $\sigma_j \ge (1-\frac{1}{2^j})\sigma_{k-1}$; clearly true for $j = k-1$. Now $(1-\frac{1}{2^j})\sigma_{k-1} \le \sigma_j \le \sigma_{k-1}$ obviously holds through. (Use $\cos\xi \ge 1-\frac{\xi^2}{2}$, $\sin\xi \le \xi$, and the assumptions on σ.)

Thus $|\sigma_o| \le \frac{8}{5}2^{-4c-4k-5}y^5$; $|\tau_o| \le \frac{8}{5}2^{-4c-4k-5}y^4$. (The latter inequality is immediate if $k = 0$; if $k > 0$, $|\tau_o| \le 2^{-2c}s_1\sigma_1 \le 2^{-2c}s_1\sigma_{k-1}$; use $s_1 \le y$, $y < 2$, $c \ge 1$.)

Now if $k > \frac{h-1}{4} - c = \frac{h-5}{4} - \bar{c}$ then $k > \frac{h+\log_2\frac{8}{5}-1}{4} - c$ (as $\log_2\frac{8}{5} < 1$); using $y < 2$, we have $|\tau_o| < \frac{1}{2^h}$, $|\sigma_o| < \frac{1}{2^{h-1}}$.

This is sufficient for τ_o, and so it is for σ_o in cases where a right-shift will follow (the leftmost bit is set). But when no right-shift follows we have $\frac{1}{2^c} \le x < \arcsin\frac{1}{2^c}$, $1 \le y < 2^c\arcsin\frac{1}{2^c}$; since $c \ge 1$, we have in any case $y < 2\arcsin\frac{1}{2} = \frac{\pi}{3}$. Then $|\sigma_o| < \frac{8}{5}2^{-h-4}(\frac{\pi}{3})^5 < 1.0075\cdot 2^{-h-3} < \frac{1}{2^h}$.

If $c = 1$ then $y \le \frac{\pi}{2}$, so we may choose $k > \frac{h-6}{4}$, which we may evidently do for $c > 1$, too.

So far, everything was for illimited-precision arithmetic. For our arithmetic, we have to show two things: that numbers fit in, and that error cumulation is acceptable. The first follows from the fact that, for arguments in the standard domain, they themselves and the results fit in, and the intermediate results decrease monotonously. The second follows from the fact that errors get "contracted", as it is to be seen in a "σ_j, τ_j" way.

sinh-cosh

We point out only the differences.

General definitions are analogous except for "+" instead of "-".

For σ_k, τ_k, σ_{k-1}, we have the same main terms, but (having no alternation) we have no strict single-term inequalities but must handle terms of "smaller order" in the obvious way.

What we want to hold through is $0 \le \sigma_j \le \exp \frac{y}{2^{c+j}}\,\sigma_{k-1}$; no special lower bound is needed now.

The expression for τ_j has opposite sign; that for σ_j is

$$\sigma_j = \cosh \frac{y}{2^{c+j+1}}\,\sigma_{j+1} + (2\sinh\frac{y}{2^{c+j+1}} - 2^{-c-j}\sigma_{j+1})(\sinh\frac{y}{2^{c+j+2}} - 2^{-c-j-3}\sigma_{j+2})\sigma_{j+2}.$$

So

$$0 \le \sigma_j \le (\tfrac{1}{2}(\exp\frac{y}{2^{c+j}}+1)+(\exp\frac{y}{2^{c+j+1}}-1)\sinh\frac{y}{2^{c+j+1}})\,\sigma_{k-1} =$$

$$(\tfrac{1}{2}(\exp\frac{y}{2^{c+1}}+1)+\tfrac{1}{2}(\exp\frac{y}{2^{c+j}}-1)-\sinh\frac{y}{2^{c+j+1}})\,\sigma_{k-1} =$$

$$(\exp\frac{y}{2^{c+j}}-\sinh\frac{y}{2^{c+j+1}})\,\sigma_{k-1}.$$

We are through; for j = 0 we need the fuller form

$$0 \le \sigma_o \le (\exp\frac{y}{2^c} - \sinh\frac{y}{2^{c+1}})\,\sigma_{k-1}.$$

Thus

$$|\sigma_o| \le (\exp\frac{y}{2^c} - \sinh\frac{y}{2^{c+1}})(\tfrac{8}{5}\,2^{-4c-4k-5}y^5+\text{small});$$

$$|\tau_o| \le (\exp\frac{y}{2^c} - 1)(\tfrac{8}{5}\,2^{-5c-4k-5}y^5+\text{small}) \text{ if } k > 0.$$

Since $\frac{y}{2^c} < \log 2$, the first factors are less than $2 - \frac{\sqrt{2}}{4}$ and 1, resp., so the "small", initially negligible as compared with the main error term, can't even be doubled during computation; forget of it.

If c = 0 then $y < \log 2$; we may choose $k > \frac{h-7}{4}$ to have $|\sigma_o| < \frac{1}{2^h}$, $|\tau_o| < \frac{1}{2^h}$. For c > 0, take $k > \frac{h-5}{4} - c$; then, since $y < 1$,

$|\tau_o| < \frac{1}{2^h}$, and, since $2 - \frac{\sqrt{2}}{4} < 1.65$, $|\sigma_o| < 1.32\cdot 2^{-4c-4k-4}$, the same k gives $|\sigma_o| < \frac{1}{2^{h-1}}$; enough if a right-shift will follow. If no right-shift follows, $y < 2^c \operatorname{arsinh}\frac{1}{2^c} = 2^c \log(\frac{1}{2^c} + \sqrt{\frac{1}{2^{2c}}+1})$; substituting this into the estimate, we have $|\sigma_o| < \frac{1}{2^h}$.
Taken together both $k > \frac{h-7}{4}$ and $k > \frac{h-5}{4} - c$ are good choices, for all c.

As to limited-precision arithmetic, the argumentation goes as with trg, except for "increase" instead of "decrease", and, albeit no error contraction, there is at worst a very limited dilatation: even the oldest error cannot grow larger than $2 - \frac{\sqrt{2}}{4}$ times its original size. (But an m sufficient for trg might happen to be a bit narrow for hyp.)

arctan

Define

$$s_o := x \;;\quad s_{j+1} := \frac{-1+\sqrt{1+s_j^2}}{s_j}.$$

We have

$$\arctan x = 2^j \arctan s_j$$

(by iterating the relation $\arctan x = 2 \arctan \frac{-1+\sqrt{1+x^2}}{x}$);

and

$$x = s_o \ge 2s_1 \ge \ldots \ge 2^j s_j \ge \ldots \ge \arctan x$$

(which is straightforward from the definition and from $s_j \ge \arctan s_j$).

What we need is $2^c \arctan x$.

We approximate this by $2^{c+k} s_k (1 - \frac{s_k^2}{3})$.

For the error, σ, we have

$$|\sigma| = |2^c \arctan x - 2^{c+k} s_k (1 - \tfrac{s_k^2}{3})| = 2^{c+k} |\arctan s_k - (s_k - \tfrac{s_k^3}{3})|$$

$$\leq 2^{c+k} \frac{s_k^5}{5} \leq \frac{8}{5} 2^{-4c-4k-3} y^5 \quad \text{(because of } s_k \leq 2^{-k} x = 2^{-c-k} y\text{)}.$$

If $c = 0$ then $y = 1$, so $k > \frac{h-3}{4}$ is enough. If $c \geq 1$ then, by $y < 2$, $k > \frac{h+1}{4} - c$ secures $|\sigma| < \frac{1}{2^{h-1}}$; enough if a right-shift follows. If no right-shift then $1 \leq y < 2^c \tan \frac{1}{2^c}$, which is greatest for $c = 1$; $2 \tan \frac{1}{2} < 1.1$; $|\sigma| < 1.3 \cdot 2^{-h-3} < \frac{1}{2^h}$.

Thus both $k > \frac{h-3}{4}$ and $k > \frac{h+1}{4} - c$ are good for all c.

Computing $2^{c+k} s_k$:

$2^c s_o$ is $2^c x = y$; $y \in [1, 2)$.

Step from $2^{c+j} s_j$ to $2^{c+j+1} s_{j+1}$:

Write

$$p := 2^{c+j} s_j \; ; \; q := 2^{c+j+1} s_{j+1} = 2^{c+j+1} \frac{-1+\sqrt{1+s_j^2}}{s_j}.$$

Compute q from p.

We have

$$(1 - 2^{-2c-2j-2} q^2) p - q = 0.$$

Decreasing q, the left-hand side increases. We know $q \leq p$.

Define $q_n := [2^n q] \div 2^n$, that is, the number constituted by the first n+1 bits of q ; q_{-1} is 0.

Suppose the first n bits of q have been found. The condition for the next bit being a 1 is

$$(1 - 2^{-2c-2j-2} (q_{n-1} + \frac{1}{2^n})^2)\, p - (q_{n-1} + \frac{1}{2^n}) \leq 0,$$

that is,

$$2^{-n} + 2^{-2c-2j-n-1} q_{n-1} p + 2^{-2c-2j-2n-2} p \leq$$

$$(1 - 2^{-2c-2j-2} q_{n-1}^2)\, p - q_{n-1}.$$

Let's estimate the right-hand side.

$$q_{n-1} = q - \frac{\varepsilon}{2^{n-1}} \text{ with an } \varepsilon \in [0, 1) \; ;$$

$$\text{rhs} = \underbrace{(1 - 2^{-2c-2j-2} q^2) p - q}_{= 0} + 2^{-n+1} (1 + 2^{-2c-2j-1} (q - 2^{-n} \varepsilon) p) \varepsilon.$$

This is nonnegative and $< 2^{-(n-1)} \frac{3}{2}$. (For $c = 0$: $p \leq 1$, $q < 1$; for $c > 0$: $p < 2$, $q < 2$.)

So is the left-hand side, too.

Multiply the condition by 2^{n-1} :

$$2^{-1} + 2^{-2c-2j-2} q_{n-1} p + 2^{-2c-2j-n-3} p \leq$$

$$2^{n-1} ((1 - 2^{-2c-2j-2} q_{n-1}^2) p - q_{n-1}).$$

Everything fits in.

Now write

$$S_n := 2^{n-1}((1-2^{-2c-2j-2}q_{n-1}^2)p-q_{n-1}) ;$$

$$T_n := 2^{-1} + 2^{-2c-2j-2}q_{n-1}p ;$$

$$R_n := 2^{-2c-2j-n-3}p .$$

For n = 0 we have $S_o = 2^{-1}p$; $T_o = 2^{-1}$; $R_o = 2^{-2c-2j-3}p$.

The condition can be written

$$T_n + R_n \leq S_n .$$

If it is satisfied - the next bit is 1 - then $S_{n+1} = 2(S_n-(T_n+R_n))$; $T_{n+1} = T_n + 2R_n$; $R_{n+1} = 2^{-1}R_n$. If it is not - the next bit is 0 - then $S_{n+1} = 2S_n$; $T_{n+1} = T_n$; $R_{n+1} = 2^{-1}R_n$.

Do this m times - for n = 0,1,...,m-1 -, and collect the bits in Q; this will be q.

From this, the suggested computation follows by obvious simplifications.

It is clear from the construction that we have as many good bits as bits (in the nonrounding sense) except the initial right-shift on S may lose a bit. [This could be avoided on most equipments; e.g. on an Am 2901 by coupling with the Q register. Moreover, a valid $(m+1)^{st}$ bit could be forced out. We didn't do so, not being pressed for m.]

[The mechanism put to operation is essentially the same as the square-root setup; it applies quite generally to

quadratically defined implicit equations. Note, in particular, that the qp products are built without a separate multiplication.]

artanh

Now $s_{j+1} := \frac{1-\sqrt{1-s_j^2}}{s_j}$, again artanh x = 2^j artanh s_j, but $x = s_o \leq 2s_1 \leq \ldots \leq 2^j s_j \leq \ldots \leq$ artanh x.

We approximate by $2^{c+k}s_k(+\frac{s_k^2}{3})$.

$$|\sigma| = 2^{c+k}(\frac{s_k^5}{5} + \frac{s_k^7}{7} + \ldots) = 2^{c+k}\frac{s_k^5}{5}(1+5(\frac{s_k^2}{7} + \frac{s_k^4}{9} + \ldots))$$

$$\leq 2^{c+k}\frac{s_k^5}{5}(1+\frac{5}{7}(s_k^2+s_k^4+\ldots)) = 2^{c+k}\frac{s_k^5}{5}(1+\frac{5}{7}\frac{s_k^2}{1-s_k^2}) ;$$

$s_k \leq 2^{-k}$ artanh x .

c = 1:

$x \leq \frac{1}{3}$; artanh $x \leq \frac{\log 2}{2}$; $|\sigma| \leq 1.124 \cdot 2^{-4k-9}$;

$k > \frac{h-9}{4}$ is good.

c > 1:

artanh $x = x + \frac{1}{3}x^3 + \frac{1}{5}x^5 + \ldots \leq x(1+\frac{1}{3}(x^2+x^4+\ldots)) =$

$x(1+\frac{1}{3}\frac{x^2}{1-x^2})$;

$x = 2^{-c}y$; $y < 1$; $c \geq 2$; $s_k \leq 2^{-c-k}(1 + \frac{1}{3}\frac{\frac{1}{16}}{1-\frac{1}{16}}) =$

$\frac{46}{45} \cdot 2^{-c-k}$; $|\sigma| \leq 1.875 \cdot 2^{-4c-4k-3}$; $k > \frac{h-3}{4} - c$ is

good; and for c = 1 too.

Computing $2^{c+k}s_k$:

The relation to be solved is

$$(1 + 2^{-2c-2j-2}q^2)p - q = 0 .$$

Decreasing q, the left-hand side increases *if* $2qp < 2^{2c+2j+2}$, which is satisfied ($p,q < 2$, $c \geq 1$). We know $q \geq p$.

The condition is

$$2^{-1} - 2^{-2c-2j-2}q_{n-1}p - 2^{-2c-2j-n-3}p \leq$$

$$2^{n-1}\ ((1+2^{-2c-2j-2}q_{n-1}^2)\ p-q_{n-1}).$$

[In this case, we could shift the condition one step more to the left. (To have this, rhs < 1 would not be enough; make sure that it is so much less that its representation will fit in.) No need to do so.]

Thus the only difference is that R_n has the negative of its previous definition, which means simply that R has to be initialized negatively.

*

The case of a zero argument has not been discussed. Yet it is clear that a zero may run through the algorithms (with any c), yielding the right results.

3.3. EPIC VS. DRAMATIC

Dramatic is somewhat shorter than Epic, and needs no "Horner constants".

As to speed, for h large, Dramatic is no match for Epic, the first requiring O(h) steps, the second, evidently, $O(\frac{h}{\log h})$ steps for direct functions (with inverse functions, the balance is better). Even for h = 25 k may be as large as 5-6, to be compared with the 4-5 "Horner" steps, where a "D" step takes at about the same number of instructions as a "Horner" step, an "I" step more than three times that much. For such a size, however, Dramatic may be competitive where memory accesses take much more than register operations (as in a microcoded "peripheral" device) or where shortness is speed in itself (e.g., a large virtual system with a small core store, or a slow system with a small high-speed device), especially if sin and cos are often needed simultaneously. For small h (e.g. for graphic routines with 10 or 12 bit precision), Dramatic may well be the better choice. In such a case an even simpler variant can be envisaged: Obviously "D" can be initialized S ← y, T ← 1 and "I" can be terminated by $S = s_k$ itself ("one-term Taylor approximation") if only k is large enough - we need then k of order $\frac{h}{2}$ (instead of $\frac{h}{4}$ now). Fixed-point arithmetic (not discussed here) can be handled the same way (and even simpler).

4. CONVERSIONS

Float and *Fix* convert from/to two- or four-byte two's-complement signed integers. Float rounds, Fix truncates in absolute value. Fix produces :8000|:80000000 ("nil") if the value cannot be fitted.

Get-Exponent, Set-Exponent handle the exponent part as a two-byte two's-complement signed integer without the "excess 128" (so the exponent in the floating representation is 128 + n).

*

Character ("formatted") conversions use "soft" routines. Their coding is self-explanatory; apparent intricacies are due to the fear of losing precision on input, to the need of rounding to the required number of digits on output.
- For details, see "KIBE, Character Handling", Budapest 1980, Sect. 5.

5. SOFT-CODED VARIANTS

The main difference is that ordinary random-access memory is used for "registers", so there is space enough. R and P take six bytes each, one for S, one for E, and four for F (the last one cleared at start); Q takes four bytes. R is not identified with any "destination"; the result gets copied to its destination location. A second operand goes to P, an only operand goes to R instead of Q; roles of P and Q in square root are interchanged. There is no need for T with division.

Even where there *is* floating-point hardware, especially if attached as a "peripheral" device, Epic functions are not always entirely microcoded but conceived as an interplay of hardware mobilizing software to call hardware: the microcode prepares computation, then induces the "host" processor to execute a piece of software (that contains floating (and non-floating) instructions), and regains control in order to conclude. - In spite of the diversity of control techniques, the algorithms themselves are unvaried.

Soft-coding being exposed to interruption even in the midst of a "floating instruction", re-entrancy has had to be cared for. - Interpretive techniques are widely used. [The loss in speed is negligible, the gain in space is enormous.]

6. CHARACTERISTICS

All this takes something like 2 Kbytes if soft-coded on an 8-bit microprocessor (less than that on a more clever device). Microcoding takes some 200 (48 bit) microinstructions for the basic operations, to which add at about 450 soft bytes for functions; fully microcoded standard functions take about another 200 microinstructions.

Multiplication time ranges from cca. 12 µsec on a bit-slice processor like Am 2901 to cca. 8 msec on Intel 8080 (TPA 70: cca. 0.3 msec, Motorola 6800: cca. 1 msec), division is slightly more, addition-subtraction somewhat faster, square root takes at about 2.5 multiplication times in soft, about 6 multiplication times in hard [multiplication being enhanced by the single-instruction cycle]; standard functions can be estimated accordingly.

Accuracy has been spoken of.

INDEX OF OPERATIONS

OTHER TITLES IN THE SERIES:

KALAP, a Batch Monitor System, 1978.

KIBE, Character Handling, 1980.

MOTOR, a Mini-Micro Blow-Up, 1983. (First printing: 1979.)

Some Mathematics for Evaluation and Design of Collision-Resolving Strategies in Networks, 1983. (First printing: 1981.)

LÓHALÁL, a Chaotic Quicknet, 1983.

IN PREPARATION:

KOTOR, a File-Server System.

TALMUD, an Interpretive Technique, and KETHAB, a Way of writing Code for It. (A streamlining of the 1973 version.)

LASSO, or Quick Khachiyan.

Hash-Cash.