This paper discusses a FORTRAN *subprogram library developed primarily to support extended-precision floating-point arithmetic. The general strategy, which makes limited use of guard digits, is developed to achieve high accuracy with reasonable execution time and storage space.*

In addition to describing some previously unpublished algorithms, the authors present subprograms for simulating extended-precision arithmetic and for input and output conversion.

FORTRAN extended-precision library

by H. Kuki and J. Ascoly

This paper discusses an extension to the traditional FORTRAN subprogram library to provide higher-level language support for extended-precision floating-point arithmetic.[1] The library includes extended-precision arithmetic simulators, input/output conversion programs, and explicitly and implicitly called extended-precision mathematical subprograms.

Extended-precision arithmetic simulation is provided to satisfy two requirements. Computing systems having extended-precision instructions normally do not have a divide instruction, and the simulator performs this operation. In computers not having extended-precision instructions, the complete set of such instructions is simulated.

A routine is provided for use in base conversion of input decimal numbers into an internally usable form, including the conversion of up to 35 decimal digits of input into the extended-precision hexadecimal form. Another routine handles output conversions, including conversion of an extended-precision number to a decimal number of up to 35 digits. This routine handles the conversion and formatting of the print field. In the cases of both routines, the coding for handling extended precision can be detached, and the remainder can be used for the standard conversions.

The extended-precision mathematical functions include square-root, logarithm, exponential, trigonometric, inverse trigonometric, hyper-

bolic, and standard complex-valued functions. They also include the three power routines, $A^{**}J$, $A^{**}B$, and $(A + Bi)^{**}J$, and the complex multiply/divide subroutine.

In this paper, we first consider accuracy goals. We then describe the extended-precision simulation and the input/output conversion routines. We finally discuss the mathematical functions, including several previously unpublished algorithms.

Accuracy goals

accuracy in general

Demand for higher quality in standard mathematical libraries has been building. Programs in the basic libraries have been subjected to thorough scrutiny in recent years[2] and are expected by users to achieve maximum accuracy.

Accuracy goals may be considered at two levels. At the first, we regard the given argument value as exact, and aim at producing an answer value that is the nearest in the given precision to the exact infinite-precision answer. Because this is the greatest accuracy that can be attained with a given number of places, we shall call it last digit accuracy. In most instances, this goal can be attained only by carrying out parts of computations in higher than the working precision of the library, especially when the relative accuracy of the result is very sensitive to the accuracy of the argument. At the second level, the fact that arguments for a subroutine have suffered through prior computations or conversions and are subject to at least minor round-off errors is taken into account. This input indeterminacy may be magnified several hundred times by the mapping of the mathematical function to produce a substantial relative error in the result. In such cases, the ability of a subroutine to attain last digit accuracy for uncontaminated arguments loses much of its significance. Instead, accuracy in the result commensurate with the effect of the minimal round-off error in the arguments would seem a reasonable goal.[3]

Unfortunately, relaxation of the accuracy goal to the second level tends to compromise users' confidence in the library.[4,5] For this reason, it is worthwhile to aim at last digit accuracy so long as the cost involved is reasonable. Moreover, the added accuracy helps to limit the accumulation of round-off errors, improving the probability of successful computation. As a simple example in which round-off errors can cause trouble, consider the identity:

$$\arccos\left[\cos\left(\frac{\pi}{3}\right)\cos\left(\frac{2\pi}{3}\right) - \sin\left(\frac{\pi}{3}\right)\sin\left(\frac{2\pi}{3}\right)\right] = \pi$$

Round-off errors can cause the quantity within the brackets to be less than -1, making it unacceptable as an argument for arccosine.

Another desirable goal is to obtain exact results whenever such are attainable or meaningful. An argument that happens to be an integer probably does not contain any error. If the corresponding function value is also an integer, it is helpful to produce such a value exactly. Also, exact conversion back and forth of integral quantities up to a certain size is desirable. In the realm of floating-point computation, such can not be attained without use of guard digits (in which additional digits are used for intermediate calculations) and true rounding (in the sense of producing the machine representable number that is the nearest to the exact answer).

conversion accuracy

Conversion between two number systems with incommensurable bases is inexact except for a small subset of special numbers. For most applications, the ideal conversion is a rounded one. Denote by $S(\beta, n)$ and $S(\delta, m)$ the sets of all floating-point numbers exactly expressible in the n-digit base β system and the m-digit base δ system, respectively. Here, for simplicity, we do not impose any bounds for the exponent range. A conversion between these systems is a mapping T from $S(\beta, n)$ into $S(\delta, m)$. A rounded conversion maps every $x \in S(\beta, n)$ to a $Tx \in S(\delta, m)$ so that

$$|x - Tx| = \min_{y \in S(\delta, m)} |x - y|$$

For the most part, such an element Tx is uniquely determined in $S(\delta, m)$. In exceptional cases, x lies exactly half way between two consecutive numbers in $S(\delta, m)$. In such cases, we would not insist on any particular algorithm for choice of either of the two neighboring values.

Due to the accumulation of round-off errors, an always correctly rounded conversion is not attainable using finite-precision arithmetic. In fact, if an input conversion is carried out using the machine arithmetic for the system $S(\delta, m)$, at least the last digit value of the result is in doubt. If $\delta = 16$, errors of up to 15 in the last digit unit can exist. Comparable errors are also introduced in the output conversion. A comparative study of the accuracy of conversions of various System/360 language processors[6] confirms this with one notable exception. The conversion by the System/360 Assembler F program attains rounded results almost always. One can approach the accuracy of an always correctly rounded conversion only by use of guard digit computation.

Traditionally users are bothered by outputs such as 0.9999998 meaning 1.0, and there has always been a strong demand for accurate conversions.[7] A virtually rounded conversion such as that provided by the Assembler F eliminates such grievances.

Awareness of the importance of rounded conversions was furthered by recent studies of D. Matula.[8,9] A conversion mapping T from

$S(\beta, n)$ to $S(\delta, m)$ is said to be "onto" if every element y in $S(\delta, m)$ is reachable as Tx by some element x of $S(\beta, n)$. It is called "one-to-one," if $x_1 \neq x_2$ means that $Tx_1 \neq Tx_2$. T and another mapping W in the opposite direction, $S(\delta, m) \rightarrow S(\beta, n)$, constitute a pair of "recoverable conversions" if $W \cdot T$ is the identity mapping of $S(\beta, n)$ onto itself. Then the following hold true.

- Given β, δ, and m, a rounded conversion T is an "onto" conversion if n is sufficiently large.
- Given β, δ, and n, a rounded conversion T is a "one-to-one" conversion if m is sufficiently large.
- Given β, δ, and n, a pair of rounded conversions (T, W) constitute recoverable conversions if m is sufficiently large.

More specifically, between the decimal system and the hexadecimal system, we can say that:

- For input conversion, we have $\beta = 10$ and $\delta = 16$. For $m = 6$, 14, or 28, the minimal values of n for which a rounded conversion is onto are $n = 9$, 18, or 35, respectively.
- For output conversion, we have $\beta = 16$ and $\delta = 10$. For $n = 6$, 14, or 28, the minimal values of m for which a rounded conversion is one-to-one are $m = 9$, 18, or 35, respectively.
- For an out-and-then-in pair of rounded conversions, the minimal decimal precision that allows recovery of internal numbers is 9, 18, or 35, depending on the hexadecimal precision 6, 14, or 28 of the internal number system.

The conversion modules of this library aim at attaining these goals of onto, one-to-one, and recoverable conversions without significant cost in speed or storage requirements.

guard digit computation

As was stated earlier, it is generally not possible to attain last digit accuracy without resorting to extra-precision intermediate computations. Suppose δx stands for the relative error of representing a quantity x by the nearest number in a floating-point system with n-bit precision. Then $0 \leqq |\delta x| \leqq 2^{-n}$. This indeterminacy in the argument x is reflected in the relative indeterminacy δy of the function value $y = f(x)$ in the following fashion:

$$|\delta y| = K(f, x)\, |\delta x|$$

where

$$K(f, x) = \frac{x}{y} \frac{df}{dx}$$

$K(f, x)$ is greater or smaller than 1 depending on the function f and the range in which x is found. Computation of the function value $f(x)$ without use of extra-precision arithmetic can not lead to

accuracy better than $K(f, x)2^{-n}$ in general.[5] In practice, the result is usually somewhat worse.

To improve the accuracy of a function subroutine, it may not be necessary to use extra-precision computation extensively. Direct computation of a mathematical function usually requires two stages, the reduction stage and the approximation in the reduced range. The first stage is to decompose the argument x as $x = P(x_j, x_\epsilon)$ so that the function value $f(x)$ can be written as $f(x) = Q(f(x_j), f(x_\epsilon))$ where P and Q are simple algebraic combinations; x_j is from a set of base values $\{x_j\}$ of x for which $f(x_j)$ are either trivially found or economically tabulatable; and accurate computation of $f(x_\epsilon)$ can be accomplished economically and stably.

The second stage consists of computing $f(x_\epsilon)$ and finally combining $f(x_\epsilon)$ and $f(x_j)$ to form $Q(f(x_j), f(x_\epsilon))$.

As an example, the sine function can be decomposed as follows: Let $x = (4n + j)(\pi/2) + x_\epsilon$ where $j = 0, 1, 2$, or 3 and $|x_\epsilon| \leqq \pi/4$. Then, writing x_{4n+j} for $(4n + j)(\pi/2)$, we have

$$\sin(x) = \sin(x_{4n+j}) \cos(x_\epsilon) + \cos(x_{4n+j}) \sin(x_\epsilon)$$

where

$$\begin{aligned}
\sin(x_{4n}) &= 0, & \cos(x_{4n}) &= 1 \\
\sin(x_{4n+1}) &= 1, & \cos(x_{4n+1}) &= 0 \\
\sin(x_{4n+2}) &= 0, & \cos(x_{4n+2}) &= -1 \\
\sin(x_{4n+3}) &= -1, & \cos(x_{4n+3}) &= 0
\end{aligned}$$

As is seen in this example, the accuracy of $f(x)$ depends on the accuracy of x_ϵ. The accuracy of x_ϵ is determined by the inherent error in x and the round-off error incurred during the reduction process. If the reduction is carried out in the working precision of the subroutine, the generated round-off error in x_ϵ is of the same order of magnitude as the indeterminacy of x in the precision. Both errors become periodically huge relative to the magnitude of x_ϵ. Here limited use of extra-precision arithmetic at the reduction stage can preserve the full accuracy of x_ϵ, which will be reflected in the final accuracy since polynomial approximations of $\sin(x_\epsilon)$ and $\cos(x_\epsilon)$ in the range $|x_\epsilon| \leqq (\pi/4)$ are stable processes.

In many applications, the function value $f(x_j)$ corresponding to an appropriately chosen base value x_j of the argument x constitutes a dominant component of the answer $f(x)$, whereas $f(x_\epsilon)$ of the reduced argument can be regarded as a perturbation part. For example, if $f(x) = e^x$ and $x = x_j + x_\epsilon$, then $f(x) = f(x_j)f(x_\epsilon)$. If we write $f(x_\epsilon) = 1 + g(x_\epsilon)$, then $f(x) = f(x_j) + f(x_j)g(x_\epsilon)$. If $|x_\epsilon| \ll 1$, then

$g(x_\epsilon) \ll 1$, and therefore $f(x) \simeq f(x_i)$ and $f(x_i)g(x_\epsilon)$ can be regarded as a perturbation part. If the constants $f(x_i)$ are given with extra accuracy (either it is a short but exact quantity or else it is encoded with guard digits), then even if $g(x_\epsilon)$ is only determined up to the base precision, $f(x_i)$ and $g(x_\epsilon)$ together determine $f(x)$ to higher than the base precision. This technique is useful when guard digits of $f(x)$ are needed for further computations or when the final rounding of $f(x)$ is desired.

Higher accuracy is generally attainable through use of a longer approximation formula and higher-precision arithmetic. Of the two, the former is relatively inexpensive, since raising the degree of a polynomial or rational approximation by one raises accuracy quite substantially. On the other hand, to carry out such an approximation in a higher precision may be expensive on some computers. This is particularly so if the base precision is the highest provided by the computer. Then higher-precision calculations require simulated arithmetic. This is one reason why guard digit computation was not applied to double-precision libraries until recently. However, with judicious application of the technique described above, extra accuracy can often be attained at minor or no additional cost. This technique is used extensively in the extended-precision library, in particular, most heavily in our algorithm for the comprehensive exponential/logarithm routine.

Extended-precision simulation

When an extended-precision instruction is encountered that is not in the set of instructions of the computer, an interruption is triggered and control is transferred to the simulator. For systems having extended-precision instructions, one simulator module is used for the only missing instruction, the divide instruction; for other models, another module simulates all extended-precision arithmetic instructions, which are listed in Table 1. Simulations of MXR and DXR by the second module require, in their turn, execution of AXR, SXR, and MXDR, and these are carried out by recursive entries to the simulator itself. Simulation of instructions other than DXR yields identical results to those of the hardware, including the treatment of the condition code and possible exceptional conditions.

The simulator is divided into three parts—prologue, main arithmetic part, and epilogue. In the prologue section, the operands are scaled to neutral exponents to ensure an exception-free computation. Simulation is carried out for the scaled operands, and the result is scaled back to the proper exponent in the epilogue section, where exceptional conditions, if any, are detected and reported.

Simulation is carried out primarily using the floating-point arithmetic instructions with occasional recourse to fixed-point arith-

Table 1 Extended-precision instructions

Name	*Mnemonic*	*Type*
ADD NORMALIZED (extended operands, extended result)	AXR	RR
SUBTRACT NORMALIZED (extended operands, extended result)	SXR	RR
MULTIPLY (extended operands, extended result)	MXR	RR
MULTIPLY (long operands, extended result)	MXDR	RR
MULTIPLY (long operands, extended result)	MXD	RX
LOAD ROUNDED (extended to long)	LRDR	RR
LOAD ROUNDED (long to short)	LRER	RR
DIVIDE (extended operands, extended result)	DXR	pseudo RR

metic whenever it is helpful. Floating-point arithmetic is preferred because the length of extended-precision arithmetic is exactly double that of long-precision. This means that we need deal only with two components of each operand. Use of fixed-point arithmetic would have required four rather than two. Moreover, the normalization function of the floating-point hardware can be gainfully utilized. This observation does not apply if the task is to simulate variable-length extended-precision arithmetic, or if simulation is to be carried out by the use of microprogramming.

The most involved simulation is that of division. The same algorithm is used for DXR regardless of the computer. The only difference is the manner of execution of the other extended-precision instructions needed to simulate DXR. If they are not provided in the computer itself, they are simulated. The specifications for DXR call for the exact truncated quotient. If x and y stand for the two operands, the exact truncated quotient $q = x/y$ is the unique extended-precision number that is characterized by the following relation:

$$q \cdot y \leqq x < (q + 16^{p-28}) \cdot y$$

$$\text{where } 16^{p-1} \leqq q < 16^{p} \quad \text{(I)}$$

The task is not as simple as it may seem at first glance. Let us evaluate the number of digits of the quotient x/y that need to be developed before we know definitely how to round this (high-precision) quotient to the correct 28 hexadecimal digits of q. Let X and Y be the values of the mantissas of x and y, interpreted as integers. Then,

$$16^{27} \leqq X < 16^{28}, \qquad 16^{27} \leqq Y < 16^{28}$$

The distance of the quotients for two such pairs of operands is given by

$$\delta = \left| \frac{X_1}{Y_1} - \frac{X_2}{Y_2} \right| = \frac{|X_1 Y_2 - X_2 Y_1|}{Y_1 Y_2}$$

The numerator is an integer, which can be equal to 1. For instance, take $Y_2 = X_1$, $X_2 = X_1 + 1$, $Y_1 = X_1 - 1$. Since $16^{54} \leqq Y_1 Y_2 < 16^{56}$, δ can be as small as 16^{-56}. This means that we need to develop 56 hexadecimal digits of the quotient before we know for sure the first 28 digits correctly.

A more economical two-step approach was used in the simulator. The first step carries out a three-stage division in the following manner.

Let $x = x_h \vee x_\ell$, $y = y_h \vee y_\ell$ be decompositions of x and y into the high-order parts and the low-order parts, respectively. Further, adopt the following symbol conventions:

Letters stand for quantities of 14 hexadecimal-digit precision, subscript h for the high-order, subscript ℓ for the low-order part.

$+$, $-$, $*$, $/$ signify long-precision operations.

$\oplus$, $\ominus$, $\circledast$ stand for AXR, SXR, and MXDR, respectively.

$\vee$ stands for concatenation of two long-precision components to form an extended-precision quantity.

Then we do the following:

$q_1 = (x_h/y_h) * c$ where $c = 1 - 16^{-12}$
$r_h \vee r_\ell = ((x_h \vee x_\ell) \ominus q_1 \circledast y_h) \ominus q_1 \circledast y_\ell$
$q_2 = r_h/y_h$
$s = ((r_h \vee r_\ell) \ominus q_2 \circledast y_h) - q_2 * y_\ell$
$q_3 = (s/y_h) + q_1 * 16^{-35}$ (to force a slight overestimate)
$q_h \vee q_\ell = (q_3 \oplus q_2) \oplus q_1$

The underestimate of q_1 by c is given to force the sign of q_1 and that of $q_3 \oplus q_2$ to agree. Without this precaution, the fact that AXR is carried out with only one guard digit can cause an excessive upward rounding in the sum, $q_h \vee q_\ell$. The sum of q_1, q_2, and s/y_h is within 16^{-39} of the infinite-precision quotient x/y, and therefore the truncated quotient $q_h \vee q_\ell$ is almost always equal to q, and when it fails to be so, it is equal to the next higher number.

The second step is to verify the inequality (I) by testing the sign of $x - y(q_h \vee q_\ell)$. This is carried out by a phased reduction. If the sign is nonnegative, $q_h \vee q_\ell = q$. Otherwise, the next lower number is chosen as q.

Conversion

input

The task of input conversion can be divided into two steps, digit accumulation and scaling. Basically, a datum for conversion consists of a decimal digit string and a decimal exponent. The digit string is converted into an internal integer, which is multiplied by an internal number that is equivalent to the indicated power of ten. Thus a table of powers of ten must be available to the conversion program.

If accuracy and speed of conversion are not important, a table consisting of a single entry, the constant 10, is sufficient. If accuracy and speed are important, a complete table of powers of ten is preferred. A compromise would be a two-stage table that contains every power between 10^1 and 10^{10}, and every tenth power thereafter. Use of this table requires scaling by two successive multiplications, which results in a modest loss of accuracy. Also, inclusion of negative powers improves speed and accuracy. Since our aim is to attain the full rounded accuracy, the precision of each power of ten in the table should exceed the working precision (We chose 14, 20, and 34 hexadecimal digits of precision for each power of ten for conversion to a short form, a long form, or an extended-precision number, respectively.) This means that a complete table would take up a large amount of storage space.

Fortunately, a scheme was devised to achieve accurate results efficiently using a table consisting of every sixth power of ten. Essentially, the scheme is to carry out part of the scaling while the digits are being accumulated by a shift of the decimal point. More specifically, the procedure is as follows. (Since the principle involved is the same for the short, the long, or the extended-precision conversion, an illustration is given only for the long-precision conversion.)

Given a digit string $\bar{x}$ of length d and an exponent p, our task is to obtain the closest 14 hexadecimal digit number to $x \cdot 10^p$, where x is the integer value of the string $\bar{x}$. First we limit d to 18, since there is no significant advantage in allowing longer strings for the long-form conversion. Let $p = 6p_0 + p_1$, where p_0 and p_1 are integers such that $0 \leqq p_1 \leqq 5$. Consider the string $\bar{x}$ to have been padded at the end with p_1 digits of zeros. We convert this $d + p_1$ digit string to a floating-point integer, 8 digits at a time, with the aid of the System/360 CVB (convert to binary) instruction. The result is then multiplied by the power 10^{6p_0} from the table to obtain the answer. The recursive accumulation $x \leftarrow x \cdot 10^8 + x_m$ is carried out in long form for the first two times, and, if the third iteration is necessary, it is carried out with guard digits to preserve the full accuracy. Due to the limitation on the value of d, at most three iterations are required for the digit accumulation. The scaling by 10^{6p_0} is then carried out with guard digits, followed by rounding.

The sparser the powers-of-ten table, the more digit padding becomes necessary. This increases the complexity of computation and decreases the speed. Using every sixth power of ten, the maximum number of iterations needed in the digit accumulation is two for short precision, three for long precision, and five for extended precision. For the short-form or long-form conversion, no use was made of extended-precision instructions.

The input conversion routine handles only the arithmetic of conversion (exclusive of the format scan).

Accuracy tests using random input data showed that only seven out of 71,000 cases failed to round correctly, and all of the failures were threshold cases, that is, they could have been effectively rounded either way.

output

The arithmetic task of output conversion can also be divided into two steps, scaling and digit generation. Basically, a datum for conversion consists of a hexadecimal floating-point number and a set of parameter values for scaling and formatting. By multiplying by an appropriate power of ten, the number is scaled to fall within a basic range such as $[1, 10^n)$, where n is a fixed integer. The integer part of this scaled number is converted to yield up to n leading decimal digits of output. The fraction part is multiplied by 10^n to yield an integer providing the next n digits, and so on. The scaling and formatting parameters, together with the index of the power of ten used in the scaling described above, are used to determine the decimal exponent, the position of the decimal point, and other quantities needed for editing the print line.

Output conversion is the reverse of input conversion, not only in its objectives but also in the sequence of subtasks involved. The problem of trade offs with regard to accuracy, speed, and storage requirements is also present in an analogous form for output conversion. It is no surprise, then, that the solution to this problem also takes a similar form. First, we set the value of n above to be 8, since it is again convenient to convert 8 digits at a time. Also, we wish to use the same power-of-ten table used for the input conversion. This is feasible, since we are scaling the given datum to the range $[1, 10^8)$, and the availability of every sixth power is sufficient to accomplish this scaling using only a single multiplication.

For the sake of brevity, we use as an illustration the F-conversion of a long-form datum. We are given a hexadecimal floating-point datum x, a decimal place index d, and a scale factor p, and we proceed as follows.

The first task is to determine the decimal scale of x, i.e., an integer J such that $10^{J-1} \leqq x < 10^J$. Although correct determination of J at the beginning simplifies subsequent logic flow, this task requires

a substantial amount of computing. It is better to make a rough estimate of J at the outset, and to make appropriate adjustments if this is proven wrong later. Only a few instructions are required to estimate $J = \text{floor}\,[\log_{10}(x) + 1]$ close enough to satisfy the relation:

$$10^{J-1} \leqq x < 2 \cdot 10^{J}$$

Then the true decimal scale of x is either J or $J + 1$. Next we evaluate the number N of significant digits to be developed as

$$N = N(J) = \min(d + p + J, 18)$$

N is limited to 18, since there is no significant merit in developing more digits if the datum is in long-precision form.

Next, we decompose J as $6j + k$, where $3 \leqq k \leqq 8$, and multiply x by 10^{-6j} to obtain a scaled value $y = x \cdot 10^{-6j}$. This multiplication is carried out with guard digits to produce 20 hexadecimal digits of y. Now $10^{k-1} \leqq y < 2 \cdot 10^{k}$, $3 \leqq k \leqq 8$. If we find $y \geqq 10^{k}$, then we raise k and J by one and adjust $N(J)$. Using this (corrected) k, we compute the rounding bias $0.5 \cdot 10^{k-N}$ and add it to y. If this rounding causes y to equal or exceed 10^{k}, we again adjust k, J, and N. At this time, we have the scaled and prerounded datum y and its correct decimal scale k, $3 \leqq k \leqq 9$.

Generation of decimal digits proceeds as follows. We have $10^{2} \leqq y < 2.5 \cdot 10^{8}$. The System/360 convert to decimal (CVD) instruction can convert integers in this range, so we use it to convert the integer part of y to obtain the first k digits of the decimal string. The fraction part of y, which, after absorption of the original guard digits, comprises a normal long-form number, is multiplied by 10^{8}. The integer part of the result is used to yield the next 8 decimal digits, and so on, until we have the required N digits.

Editing a FORTRAN print line requires determination of various parameter values. They are, from left to right along the print line:

- The number of leading blanks
- The sign of the number
- The number of significant digits before the decimal point
- The number of zeros before the decimal point
- The number of zeros after the decimal point
- The number of significant digits after the decimal point
- The number of trailing blanks
- The exponent field

These are determined, using the standard rule, from the values of d, p, J, and the print field width. Special care is necessary when the value of N turns out to be negative.

For G-conversion, the following interpretation was taken as to the effect of a scale factor p. First, note that the parameter d is defined

to be the number of significant digits, i.e., $N = d$. The American Standard specifies the following:

- If $0 \leqq J \leqq d$, then the sliding point F-type conversion is used. In this case, the scale factor p is ignored.
- If J is outside $[0, d]$, E-type (or D, or Q^{10}) conversion is used. In this case, the scale factor p is in effect.

Our implementation of the second case is as follows:

- First we limit p to the range $[0, d]$, i.e., force p to 0 if $p < 0$ and force p to d if $p > d$. This is done because the value of p outside $[0, d]$ has no practical use and, if allowed, could cause considerable technical difficulties.
- In order to comply with the definition of the parameter d as the precision, we place p significant digits in front of the decimal point and $d - p$ significant digits following the decimal point. This allows the printed digits (and the decimal point) to fit into the allocated $d + 1$ positions of the print field.
- The decimal exponent is adjusted to reflect the shift of the decimal point by p.

This interpretation is in conformity with the primary purpose of the G-conversion, namely an extension of the F-conversion with the aim of achieving as much precision as possible within the allocated space.

Accuracy tests using random data showed 65 cases of failure to round correctly out of a total of 71,000 tried. The worst case observed was in the conversion of an extended-precision number to 35 decimal digits where the error was 0.566 in the 35th digit.

For conversion of long-form or short-form numbers, use of extended-precision arithmetic is avoided.

Mathematical functions

choice of approximations in general

As was discussed earlier, computation of a mathematical function consists in general of two stages—the reduction stage and the approximation stage. The basic range into which the argument is reduced must be such that within this range, one has an approximation algorithm that is both efficient and stable. By efficiency, we mean economy of speed and storage requirements while attaining a predetermined accuracy goal. Our accuracy goal is to keep the maximum relative error well within the range of the last digit value of the working precision.

Among polynomial (or rational) approximations of a given degree, there is one that minimizes the maximum error in the given range.

Such an approximation is called a minimax approximation, and we seek such an approximation of the lowest degree that satisfies our accuracy goal. Of the several algorithms available to us to determine coefficients of such approximations, we used two, Maehly's second direct method and Remez' second method.[11,12] Coefficients were computed by these methods in 38 hexadecimal digits of precision before being rounded to extended precision. A multiple precision arithmetic package built by John Ehrman of Stanford Linear Accelerator Center was used in this computation.

Often the minimax technique was applied so as to obtain the minimax with a constraint. Consider for example a polynomial approximation of $\cos(x)$, $|x| \leqq \pi/4$. Let

$$P(x^2) = \sum_{i=0}^{n} a_i x^{2i}$$

be an unconstrained minimax approximation in the range with an error ϵ, i.e., $\cos(x) = P(x^2) + \epsilon$. Since $\cos(x)$, as a function of x^2, has a standard error curve, the maximum error $\epsilon_0 = \max |\epsilon|$ is attained at the end point $x^2 = 0$ of the range. This means that $a_0 = 1 \pm \epsilon_0$. If the approximation meets our goal, then ϵ_0 is less than the round-off error of fitting the constant a_0 to our working precision. This means that, by encoding the constant a_0 in the working precision, we would be introducing another error of the size ϵ_0, so that the maximum error of the computed answer is at least $2\epsilon_0$. If this has a visible effect on the answer, we wish to obtain an approximation that is exact at $x^2 = 0$ and has the smallest possible error elsewhere in the range. One way to accomplish this is as follows. Define a function $f(w)$ on $[0, \pi^2/16]$ by

$$f(w) = \begin{cases} \dfrac{\cos(\sqrt{w}) - 1}{w} & \text{if} \quad w \neq 0 \\ -\frac{1}{2} & \text{if} \quad w = 0 \end{cases}$$

Then apply the minimax technique to obtain an approximation $P(w)$ of $f(w)$ that minimizes the maximum of $E \cdot w/\cos(\sqrt{w})$, where E stands for the absolute error $E = f(w) - P(w)$. This minimax problem has a nonstandard error curve, but either one of two minimax methods we used produced acceptable results for our application. One verifies easily that $E \cdot x^2/\cos(x)$ is the relative error of the approximation $1 + x^2 P(x^2)$ of $\cos(x)$, and that this approximation is exact at $x = 0$.

Another example of a minimax problem with constraints is seen in the following. A particularly efficient form of approximation of the exponential function is

$$2^x \cong R(x) = 1 + \frac{2x \cdot P(x^2)}{Q(x^2) - x \cdot P(x^2)}$$

where P and Q are polynomials of given degrees. Efficiency of this form is due to the existence of a rapidly converging continued fraction expansion of the hyperbolic tangent function. We have

$$\frac{P(x^2)}{Q(x^2)} = \frac{R(x) - 1}{x \cdot (R(x) + 1)}$$

Hence, we define a function $f(w)$ of nonnegative argument w as

$$f(w) = \frac{2^x - 1}{x \cdot (2^x + 1)}, \qquad x = \sqrt{w},$$

and apply the minimax technique on $f(w)$ to determine the coefficients of P and Q. Let

$$f(w) = \frac{P(w)}{Q(w)} + E$$

Then the relative error of $R(x)$ as an approximation of 2^x is given by

$$1 - \frac{R(x)}{2^x} = 1 - \frac{Q + xP}{Q - xP} \cdot \frac{Q - xP - xQE}{Q + xP + xQE}$$

$$= \frac{Ex}{2}(1 + R(x))(1 + 2^{-x}) \cong \frac{x}{2}(2 + 2^x + 2^{-x})E$$

The last expression provides the weight function relative to which the E is to be minimized.

The fact that the extended-precision instruction set does not provide the divide instruction may indicate preference of polynomials over rational functions as approximations. Complete simulation of an arithmetic instruction invariably takes much longer than its equivalent in machine arithmetic would. However, when a division is called for in computing mathematical functions, the problem is often so localized that the exponent scaling of operands is not necessary. Also, accuracy requirements for division vary depending upon applications. Under these conditions, division may be simulated quite economically. On the other hand, for many functions, rational approximations tend to be much more efficient than polynomial approximations, and this economy is quite substantial when approximations of high degrees are involved. Thus we found ourselves selecting more rational approximations than we had anticipated originally.

We considered use of methods such as Pan's for economical evaluation of polynomials but in the end chose not to use them. These methods allow polynomials to be evaluated with fewer multiplications than the standard method of nested multiplication. However, algorithms based on such methods tend to show poor round-off characteristics. Moreover, a trial code revealed that they are not

particularly efficient for our applications. For one thing, the logic flow becomes more complicated than that of nested multiplication, and it requires more frequent loading and storing of register contents. Also, since the polynomials we use are those with rapidly diminishing high-degree terms, about 30 percent of the iterations in the nested multiplication can be carried out in long precision. In our example, Pan's method was no faster than the nested multiplication and it required considerably more storage space.

exponential/ logarithm subroutine

The traditional approach to computing the real-to-real exponentiation x^y is to compute exp $(y \cdot \log (x))$ by use of external subroutines for computation of the exponential and the logarithm functions. It was recognized early, however, that one can not attain very high accuracy in this way regardless of the accuracy of the two subroutines used.[13] This is due to the fact that, when the magnitude of y is large, the working precision accuracy of log (x) is not sufficient to assure the relative accuracy of x^y. Being external subroutines, both the output of the logarithm subroutine and the input to the exponential subroutine are limited to the working precision. The size of errors in x^y caused by these limitations is comparable to the effect of the argument indeterminacy discussed earlier. However, in view of the importance of this function, it is highly desirable to attain a higher accuracy. In particular, it is desirable to obtain exact results if both the base x and the exponent y are integral quantities.

In order to attain this goal, it is necessary to develop intermediate results with higher than the working precision. This was tried before, at the University of Toronto and at Argonne National Laboratory in the early 1960's, and later at the University of Chicago. It is worth noting that these efforts differed in their packaging concepts. The Toronto approach was to build into the exponential subroutine and the logarithm subroutine a facility to distinguish calls by the x^y routine from other calls and to pass on a few hidden guard digits to and from the x^y routine. The Argonne approach was to have a stand-alone x^y subroutine independent of and in addition to the exponential and logarithm subroutines. The Chicago approach was to prepare the x^y subroutine with a minor entry point for computation of the exponential function, so that if only the exponential function were needed, a shorter, separate subroutine would be loaded. The approach used in this library is to combine in a single subroutine five functions—the base e logarithm, the base 10 logarithm, the base e exponential, the base 2 exponential, and the real-to-real exponentiation.

The basic algorithm consists of two parts—that for computing the base 2 logarithm and that for computing the base 2 exponential. Other functions are trivially obtainable from these two. x^y is computed as $2^{y \cdot \log_2 (x)}$. In order to achieve our accuracy goal, we aim at computing $\log_2 (x)$ with two extra hexadecimal digits of precision. To do this economically, we need to reduce the argument sharply

so that the perturbation part can be computed in the working precision.

We write $x = 16^p 2^{-q} m$, where q is 0, 1, 2, or 3 and $0.5 \leqq m < 1$. We then choose an appropriately distributed set of base points α_n in [0.5, 1], and, for the given m, choose the closest α_n. Then, letting $z = (m - \alpha_n)/(m + \alpha_n)$, we have:

$$\log_2 (x) = 4p - q + \log_2 \alpha_n + \log_2 \left(\frac{1 + z}{1 - z}\right)$$

Here $4p - q + \log_2 (\alpha_n)$ is the dominant part, and $\log_2 [(1 + z)/(1 - z)]$ is the perturbation part. We wish to limit the latter to less than 16^{-2} in magnitude. This requires more than 129 values of α_n and their high-precision logarithmic values. Since we can not afford to carry so many constants, we proceed as follows.

We estimate $\log_2 (m)$ very crudely, and obtain three indexes $0 \leqq i \leqq 8, 0 \leqq j \leqq 3, 0 \leqq k \leqq 4$, so that $20i + 5j + k$ is the nearest integer to $-160 \cdot \log_2 (m)$. Using these indexes, we select three constants β_i, γ_j, and δ_k, where $\beta_i = [2^{-i/8}]$, $\gamma_j = [2^{-j/32}]$, and $\delta_k = [2^{-k/160}]$. Here the brackets denote rounding to the closest 17-bit quantity. These 18 constants, $\beta_i, \gamma_j, \delta_k$, and their logarithms (in 34 hexadecimal digits) are encoded in the program. Then the economically computable product $\alpha_{ijk} = \beta_i \gamma_j \delta_k$ is taken to be the base point for m. This is sufficiently close to m. In fact, the magnitude of $\log_2 [(1 + z)/(1 - z)]$ is not much bigger than $1/320$, and an approximation of the form $zP(z^2)$ in the working precision, where P stands for a polynomial of degree 5, is sufficient for our purpose. Extra precision is maintained as follows: The integer $4p - q$ and the high-order part of $\log_2 \alpha_{ijk}$ $(= \log_2 \beta_i + \log_2 \gamma_j + \log_2 \delta_k)$ comprise the dominant part of $\log_2 (x)$ and are kept in short precision. The sum of the low-order part of $\log_2 \alpha_{ijk}$ and $\log_2 [(1 + z)/(1 - z)]$ comprises the perturbation part and is kept in extended precision. Together they represent $\log_2 (x)$ within 16^{-30} in absolute error or 16^{-28} in relative error, whichever is smaller.

Having obtained $\log_2 (x)$ in such high precision, we multiply it by y with care, preserving this accuracy. Then we raise the product z to the power of 2 as follows.

First we decompose z as $z = 4p - q - r$, where $q = 0, 1, 2$, or 3, and $0 \leqq r < 1$. We find two indexes, i, j, $0 \leqq i \leqq 8, 0 \leqq j \leqq 3$, such that $4i + j$ is the integer nearest to $32r$. Using these indexes, we obtain the product $\varphi_{ij} = \beta_i \gamma_j$, where the factors β_i, γ_j are from the pool of constants described above. We obtain the reduced argument $s = -r - \log_2 \varphi_{ij}$ accurately by subtracting $\log_2 \varphi_{ij} = \log_2 \beta_i + \log_2 \gamma_j$ from $-r$ in steps. The quantity s is approximately bounded by $\pm 1/64$.

We compute 2^s by a minimax approximation of the form:

$$2^s = 1 + \frac{2sP(s^2)}{Q(s^2) - sP(s^2)}$$

where P and Q are polynomials of degree 2. Finally, we assemble the power 2^z as $16^p \cdot (2^{-q}\varphi_{i_1}) \cdot 2^s$ and in the process apply the rounding.

Tests show that the results of exponentiation x^y are virtually always correctly rounded except for cases where x is very close to 1.0 and y is very large. In particular, if x is an integer less than 1000, and y is an integer such that $x^y < 16^{28}$, then the result is always exact. This statement is also expected to apply when x exceeds 1000.

A long precision implementation of this algorithm was also coded.[14] It is to be noted that the cost of argument reduction in this algorithm is substantial and is largely independent of the working precision. A less sharp reduction, on the other hand, would have required more expensive approximations in the basic ranges. For the extended-precision version, the cost of reduction is amply offset by the economy in basic approximations. For the long-precision version, the latter is not quite enough to cover the former. For example, computation of DEXP and DLOG according to this algorithm is slower by 15 to 25 percent than the subroutine in the current FORTRAN library, and this is the price one pays for the improved accuracy of x^y. Computation of *DX**DY* according to our algorithm is still slightly faster than that of the current library. It would be prohibitively expensive to apply this particular algorithm to the short-precision library.

square-root subroutine

Despite the lack of an extended-precision divide instruction, division iterations in the manner of the Newton-Raphson refinements turned out to be more efficient than several multiplicative iteration formulas we have tried. This is partly due to the fact that we can still use short- and long-precision divisions for all but the final iteration, and also to our choice of the final iteration formula that requires only a very rough simulation of the extended-precision division.

By scaling, we reduce the problem to the case where $x = 16^{32} \cdot m$, $1/16 \leqq m < 1$. This particular scaling is used to avoid possible intermediate underflows. The first approximation y_0 to $\sqrt{x}$ is computed as,

$$y_0 = 16^{16} \cdot \left(1.807018 - \frac{1.576942}{0.9540356 + m}\right)$$

These coefficients were determined to minimize the relative error of the approximation while being exact at $m = 1$. The maximum relative error of y_0 over the range $1/16 \leqq m \leqq 1$ is $2^{-5.48}$.

We Apply Newton-Raphson iteration three times—twice in short precision and once in long precision:

$$y_i = \frac{1}{2}\left(y_{i-1} + \frac{x}{y_{i-1}}\right) \qquad i = 1, 2, 3$$

At the end of the third iteration, the relative error ϵ_3 of y_3 is at most 2^{-41}.

The final iteration has the following form[15] and is carried out in extended precision:

$$y_4 = y_3 - 2y_3 \cdot \frac{y_3^2 - x}{3y_3^2 + x}$$

By substituting $(1 + \epsilon_3)\sqrt{x}$ for y_3 in this formula, one sees that the relative error ϵ_4 of y_4 is essentially equal to $\frac{1}{4}\epsilon_3^3$, or 2^{-125} in this case.

Note that the right-hand term of the formula is only a correctional term to y_3 and is limited by 2^{-41} in relative magnitude. Therefore a rough simulated division that takes only 1.5 times MXR time (mentioned above) is sufficient for computing this term. In the process of combining this term with y_3, a rounding bias is introduced to attain rounded results virtually always.

trigonometric functions

As was discussed earlier with the example of the sine function, the relative accuracy of the computed value of a trigonometric function depends largely on the care exercised in the reduction stage. For this reason, it is desirable to use an arithmetic of higher than the working precision during this stage. It is also desirable to use an identical reduction algorithm for all trigonometric functions, so that an argument is reduced to the same value in the basic range regardless of the function involved. In this way, we are able to produce function values that satisfy various trigonometric identities as accurately as the approximations used in the basic range can satisfy them.

Our task is to decompose the given argument x as $x = (\pi/2)n + r$, where n is an integer and $-\pi/4 < r \leqq \pi/4$. Given x, there is exactly one pair of values (n, r) that satisfies this relation. Upon closer examination of our algorithm, however, we find that the requirement $-\pi/4 < r \leqq \pi/4$ is not a critical one; that is, we can enlarge our basic range modestly without paying any significant penalty. On the other hand, the identity $x = (\pi/2)n + r$ must be maintained accurately. By this relaxed rule, there can occasionally be more than one choice of n for the given value of x; but once n is chosen, the remainder must be computed very accurately.

More specifically, we compute $(2/\pi)x + 0.5$ in the working precision, and take the integer part of this quantity to obtain n. Then, by a staged reduction, we subtract the product $(\pi/2)n$ from x to obtain the reduced argument r, employing in the process approximately 10 hexadecimal guard digits. Since $|r|$ can be a few percent larger

than $\pi/4$ due to round-off if $|x|$ is large, the approximation formula for the reduced argument must be chosen so as to be valid over this slightly enlarged basic range. This is done without raising the degree of approximation formula. Thus the use of extra-precision arithmetic is limited to the staged reduction $x - (\pi/2)n$. The cost for this is fairly minor.

Within the basic range, minimax approximations of the following forms are used for respective functions (P_k stands for a polynomial of degree k):

$$\sin(r) \cong r + r^3 \cdot P_{10}(r^2)$$

$$\cos(r) \cong 1 + r^2 \cdot P_{11}(r^2)$$

$$\tan(r) \cong r \cdot P_6(r^2)/P_5(r^2)$$

$$\cot(r) \cong P_5(r^2)/(r \cdot P_6(r^2))$$

Algorithms for arcsine and arccosine functions follow essentially the same logic as those used in the short- and long-precision versions of the current FORTRAN library. On the other hand, a more elaborate scheme was introduced for computation of the arctangent function in extended precision to lower the degree of approximation and also to limit the number of divisions involved. It proceeds as follows:

We choose origins θ_i, $i = 1, 2, \cdots, 7$, to be approximately $(i/16)\pi$ in such a way that $\tan \theta_i$ are exact short-form numbers, and let $\theta_0 = 0$ and $\theta_8 = \pi/2$. We define break points β_i, $i = 0, 1, \cdots, 8$, as

$$\beta_i = \tan\left(\frac{2i - 1}{32}\pi\right)$$

If $\beta_i \leqq x < \beta_{i+1}$ for $i = 1, 2, \cdots, 7$, we use the reduction:

$$\arctan(x) = \theta_i + \arctan\left(\frac{x - \tan\theta_i}{1 + x \cdot \tan\theta_i}\right)$$

If $\beta_8 \leqq x < \infty$, we use the reduction:

$$\arctan(x) = \theta_8 + \arctan\left(\frac{-1}{x}\right)$$

In either case, the quantity within the parenthesis on the right is within the basic range (β_0, β_1), i.e., less than $\tan(\pi/32)$ in magnitude. Within the basic range, a minimax approximation of the form $x + x^3 \cdot P_{11}(x^2)$ is used to evaluate $\arctan(x)$.

For arctangent of two arguments, care was exercised to avoid premature underflows and overflows, as was done in the short-form and the long-form versions of the current FORTRAN library. However, if we find, by a short-form division, that $\beta_8 \leqq |x_1/x_2| < \infty$, then we use the following reduction to eliminate one division:

$$\arctan\left(\left|\frac{x_1}{x_2}\right|\right) = \theta_8 + \arctan\left(\frac{-|x_2|}{|x_1|}\right)$$

other functions

The algorithms for the hyperbolic functions follow essentially the same logic as those used in the short-form and the long-form versions. The algorithms for the complex-valued functions were somewhat simplified in several cases by use of direct scaling. The cost of exponent scaling is independent of the precision involved, and therefore it becomes relatively minor for higher-precision computations.

A new scaling method was devised for the complex multiply and divide subroutine. Let the two operands be denoted by $a + bi$ and $c + di$. Then,

$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i, \qquad \text{and}$$

$$\frac{a + bi}{c + di} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i$$

Direct computation of these formulas can lead to premature overflows and underflows. For multiplication, this occurs only when the result approaches the overflow or the underflow threshold, whereas a loss of accuracy due to partial underflows can occur somewhat earlier. For division, overflows and underflows can occur even if the result is near 1, and the operands are well within the legitimate exponent range. Thus some form of scaling of the operands is called for.

Scaling the operands by dividing them by max ($|c|$, $|d|$), as is done in some programs, is out of the question for the extended-precision library. Moreover, this method increases roundoff errors somewhat. Scaling by adjusting the exponents of operands is acceptable, but we wish to minimize the cost of doing so. The scheme we devised was to scale only one operand instead of two as is usually done, and yet to provide the full protection. We leave the first operand $a + bi$ intact and scale the second operand $c + di$ in such a way as to counter balance the magnitude of the first operand. For division, a midcourse change of scale becomes necessary for successful computation.

First, we determine two exponents p_1, p_2 that satisfy:

$$16^{p_1 - 1} \leqq \max(|a|, |b|) < 16^{p_1} \qquad \text{and}$$

$$16^{p_2 - 1} \leqq \max(|c|, |d|) < 16^{p_2}$$

Then we choose the exponent q as $q = -3$ if $p_1 \geqq 0$, and $q = 31$ if $p_1 < 0$. We scale c and d by $16^{p_2 - q}$ and reverse the sign of d if the operation is a divide:

$$c_1 = c \cdot 16^{q-p_2}$$

$$d_1 = \begin{cases} d \cdot 16^{q-p_2} & \text{if complex multiplication} \\ -d \cdot 16^{q-p_2} & \text{if complex division} \end{cases}$$

Here, scaling is done by exponent adjustment, and if this results in underflow of one of the components, we replace the affected component by 0. Then we compute:

$$u_1 + v_1 i = (ac_1 - bd_1) + (ad_1 + bc_1)i$$

Since we have $16^{q-1} \leqq \max(|c_1|, |d_1|) < 16^q$, and also $-34 \leqq p_1 + q \leqq 60$, this calculation causes no overflow and no significant underflow. The product is obtained by scaling back $u_1 + v_1 i$ by 16^{q-p_2}.

If the quotient is our aim, then we compute the divisor $w_1 = (c_1^2 + d_1^2) \cdot 16^{-2q}$. This computation is safe since $16^{2q-2} \leqq c_1^2 + d_1^2 < 2 \cdot 16^{2q}$, and we have $16^{-2} \leqq w_1 < 2$. Now we compute the quotients $u_2 = u_1/w_1$ and $v_2 = v_1/w_1$ and scale them back to obtain the answer $u + vi$: $u = u_2 \cdot 16^{-q-p_2}$ and $v = v_2 \cdot 16^{-q-p_2}$.

At each intermediate stage, the magnitude of the result is within $(16^{-35}, 16^{63})$, safely within the exponent range of our number system. If an overflow or underflow condition is present in the answer, it is detected at the final stage of scale restoration.

There are two economical algorithms for building powers for integer power subroutines: *X**J* and *(X + Yi)**J*. One can be called the "bottom up" method, and the other the "top down" method. L. R. Turner of NASA Lewis Research Center had communicated to us that the "top down" method is somewhat more accurate than the "bottom up" method. This was confirmed by our tests, and since the former is only insignificantly slower than the latter, we have adopted the "top down" method. In the case of computing x^J, the method proceeds as follows.

Assume $x \neq 0$ and $J > 0$. Let the binary representation of J be

$$J = \sum_{i=0}^{n} g_i 2^{n-i}$$

where $g_0 = 1$ and $g_j = 0$ or 1 for $j = 1, 2, \cdots, n$. Thus $n + 1$ is the number of significant binary digits of J. We initialize $z_0 = x$. If $n = 0$, $z = z_0$ is the answer. We do the following for $j = 1, 2, \cdots, n$:

$$z_j = \begin{cases} z_{j-1}^2 & \text{if } g_j = 0 \\ z_{j-1}^2 \cdot x & \text{if } g_j = 1 \end{cases}$$

Then, at the end of iterations, $z = z_n$ is the answer.

This is "top down" since we scan the binary representation of J from the left to right while building the power. The "bottom up" method, which is used in the short-form and the long-form versions in the current FORTRAN library scans the representation of J from right to left.

It is to be noted that, either way we build up powers, the growth of round-off errors is roughly proportional to $J - 1$. Our tests also showed that between two exponents of similar magnitudes, the incidence of one's in their binary representations has little to do with accumulation of roundoff errors. For example, we did not recognize any significant difference between error statistics for x^{8191} and x^{8192}, despite the fact that computation of the former requires 24 multiplications and that of the latter 13 multiplications.

Concluding remarks

Each subroutine in the FORTRAN extended-precision library has been subjected to a set of rigorous performance tests involving thousands of random sample arguments. Master reference programs that compute these functions in 38 hexadecimal digits of precision were used to measure the accuracy of the tested programs.

Despite the elaborate care exercised in preparation of the programs in this library, these programs cannot create accuracy where there is none. The accuracy of the computed function value depends on the quality of the input argument as well as the quality of the algorithm used. Seemingly minor contamination of the input value can cause substantial relative errors. The most one can expect and all we have attempted is to minimize any error due to computational method by careful coding, which includes occasional use of guard digits. The average cost of this use of extra precision arithmetic is estimated to be about 10 percent in execution time and somewhat more in storage requirements. We believe this to be a reasonable price to pay for the extra precision of the results.

ACKNOWLEDGMENTS

The following members of the University of Chicago Computation Center have participated in the development of this library: Gary Duggan, Marcia Kastner, John Keck, Paul Kinnucan, Thomas Morgan, and Dorothy Raden.

CITED REFERENCES AND FOOTNOTES

1. This capability is provided in the IBM System/360 Models 85 and 195 and in the IBM System/370 computers. See A. Padegs, "Structural aspects of the System/360 Model 85, III, Extension to floating-point architecture" *IBM Systems Journal* **7**, No. 1, 22–29 (1968).

2. W. J. Cody, "Software for the elementary functions," Mathematical Software Symposium at Purdue University, (April 1970). Proceedings to appear.
3. H. Kuki, "Comments on the ANL evaluation of OS/360 FORTRAN Math Function Library," SHARE Secretary Distribution 169, C-4773, pp 47–53 (July 1967).
4. W. J. Cody, "Critique of the FORTRAN 1V(H) Library for the System/360," SHARE Secretary Distribution 169, C-4773, pp 3–46 (July 1967).
5. H. Kuki, "Mathematical function subprograms for basic system libraries: objectives, constraints, and trade-offs," Mathematical Software Symposium at Purdue University, (April 1970). Proceedings to appear.
6. J. R. Ehrman, "A study of floating-point conversions in some OS/360 components," SHARE Secretary Distribution 195, C-5207, (June 1969).
7. D. Warnock, "Report from SHARE FORTRAN project," SHARE Secretary Distribution 203, C-5340 (April 1970).
8. D. W. Matula, "A formalization of floating point numeric base conversion," Computer Systems Laboratory Technical Report No. 17, Washington University, St. Louis, Missouri (March 1970).
9. D. W. Matula, "In and out conversion," *Communications of the ACM* **11**, No. 1, 47–50 (January 1968).
10. Q is the FORTRAN format code that specifies conversion of up to thirty-five decimal digits to or from their hexadecimal equivalent.
11. J. F. Hart et al, *Computer approximations,* John Wiley & Sons, Inc., New York, New York (1968).
12. C. T. Fike, *Computer evaluation of mathematical functions,* Prentice-Hall, Englewood Cliffs, New Jersey (1968).
13. N. W. Clark and W. J. Cody, "Self-contained exponentiation," AFIPS Conference Proceedings, **35**, 701–706 (1969).
14. N. W. Clark, W. J. Cody, and H. Kuki, "Self-contained power routines," Mathematical Software Symposium at Purdue University, (April 1970). Proceedings to appear.
15. I. Wladawsky, private communication (1966). Also, an equivalent form, due to R. Dedekind, appears in *Survey of Numerical Analysis,* edited by J. Todd, McGraw-Hill Publishing Co., Inc., New York, p. 22 (1962).