New scalar and vector elementary functions for the IBM System/370

by Ramesh C. Agarwal
James W. Cooley
Fred G. Gustavson
James B. Shearer
Gordon Slishman
Bryant Tuckerman

Algorithms have been developed to compute short- and long-precision elementary functions: SIN, COS, TAN, COTAN, LOG, LOG10, EXP, POWER, SQRT, ATAN, ASIN, ACOS, ATAN2, and CABS, in scalar (28 functions) and vector (22 functions) mode. They have been implemented as part of the new VS FORTRAN library recently announced along with the IBM 3090 Vector Facility. These algorithms are essentially table-based algorithms. An important feature of these algorithms is that they produce bitwise-identical results on scalar and vector System/370 machines. Of these, for five functions the computed value result is always the correctly rounded value of the infinite-precision result. For the rest of the functions, the value returned is one of the two floating-point neighbors bordering the infinite-precision result, which implies exact results if they are machine-representable. For the five correctly rounded elementary functions, scalar and vector algorithms are designed independently so as to optimize performance in each case. For other functions, the bitwise-identical constraint leads to algorithms which compromise between scalar and vector performance. We have been able to design algorithms where this compromise is minimal and thus achieve very good performance on both scalar and vector implementations. For our test measurements on high-end System/370 machines, our scalar functions are always faster (sometimes by as much as a factor of 2.5) as compared to the old VS FORTRAN library. Our vector functions are usually two to three times faster than our scalar functions.

1. Introduction

• *History of the FORTRAN intrinsic functions*

When FORTRAN became widely available, there were already many subroutines [1] for elementary functions in the SHARE (IBM users' organization) library. In addition, many computer installations had their own libraries, obtained from various sources. These were written for assembly language programs but could be used with FORTRAN. One compiled and ran in separate sessions and in the run session one could use subroutines of one's own choosing.

It soon became obvious that a standard "best" set of subroutines should be included in all of IBM's versions of FORTRAN. The original FORTRAN library that came with 704/709/7090 FORTRAN was written by IBM. It was "adequate," but many programmers continued writing new and improved versions of various routines. Eventually most SHARE installations converged on a set of improved versions (mainly by W. Kahan at the University of Toronto,

W. J. Cody at the Argonne National Laboratory, and H. Kuki at the University of Chicago) and distributed them via the SHARE library. The new routines were good enough for IBM to take them over as the official versions.

When work began on System/360, Kuki was contracted to write the new library (since he had done the bulk of the then-current version). The first set of routines was available with the initial releases of System/360 FORTRAN. See [2–4] for general discussions of the problem and objectives of computing elementary functions.

In his presentation [5] at the Mini-Symposium on Elementary Functions at the SIAM National Meeting in Seattle in July 1984, titled "Software for the Elementary Functions," Cody described these subroutines:

The result was an excellent library, but still not the best that could be done, even by standards of that time. Kuki was severely handicapped by a requirement that each program had to produce identical numerical results on all machines in the family. Speed and space restrictions imposed by the smaller machines in the line, and variations in the machine architectures, meant that he could not produce programs that were optimal for any one of the machines. For example, because of timing considerations on the smaller machines, he used careful argument reduction on single-precision (32-bit) programs but not double-precision (64-bit) ones, and he minimized the use of double-precision floating point division. This penalized those doing scientific computations on larger machines to meet restrictions imposed by smaller machines intended primarily for non-scientific use. Because of the inadequacy of the double-precision programs, many installations reverted to a non-standard library containing replacement programs written by Kuki or the group at Argonne.

The first set of modifications to these routines was made at the time of the "Improved Floating Point Engineering Change" (IFPEC), which fixed some of the deficiencies (most importantly, the lack of a guard digit in long floating-point arithmetic) in the original floating-point architecture. Kuki made modifications to some of the routines to exploit the corrections (e.g., the fact that the Halve instructions now produced a normalized result), but the changes were not extensive.

When the System/360 Model 85 was in design [with extended (128-bit) precision], Kuki was asked (see [6]) to revise the library to add extended-precision functions. That version of the library was announced with FORTRAN H Extended about 1971 and has been the "standard" library ever since [6].

At the SHARE-54 meeting in Anaheim, California, in March 1980, Wang [7] (see also [8, 9]) described some inadequacies in the IBM System/360 FORTRAN IV libraries and made suggestions for their improvement. He also described some results with a new set of subroutines which overcame these difficulties. The SHARE FORTRAN Project submitted a requirement to IBM to provide the Wang routines as an alternate to the standard product library; IBM accepted the requirement and shipped the routines with VS FORTRAN Version 1 Release 2 as VALTLIB in 1982. The routines did not meet with enormous acceptance due to the trigonometric functions being 20–40% slower than the standard library; many users with small arguments stayed with the original versions, even though for large arguments the standard library was less accurate.

Scarborough [10] wrote versions of the library routines as part of the Optimization Enhancement IUP (FORTRAN Q). He made no changes to the Kuki algorithms, but changed the programs to use shorter execution paths with less preparation for error cases until the errors actually occurred. He also changed the conversion routines slightly, again to gain speed but with no perceptible change in accuracy.

In the earlier computers, memory space was quite small and computer speed was orders of magnitude higher than many users were accustomed to. Compactness of the program and accuracy were therefore the most important considerations. As succeeding generations of machines and FORTRAN revisions were produced, the practice was to be able to assure users that the new versions would yield the same results as the old ones (see [10], for example). In many cases, certain FORTRAN programs were established as the tests of acceptability of plans and designs. Therefore, and this may be folklore, succeeding generations of users requested that new subroutines give exactly the same results as old ones even when the old ones were inaccurate.

- *The need for new functions*

After some time, the requirement of complete numeric compatibility became an undesirable constraint. A compelling reason for a break with this rule came with the advent of IBM's new Vector Facility. We thought that a program using the new vector instructions to mimic the old algorithms in such a way as to produce exactly the same results would result in inefficient vector subroutines. Furthermore, since a good algorithm for a scalar subroutine is, in general, not a good algorithm for a vector subroutine, and since the previous algorithms used divide instructions, it was necessary for the sake of speed to develop new algorithms and, therefore, forego the requirement for strict compatibility with the old subroutines. However, we agreed to provide the user the ability to get the same results from the subroutines whether or not he used the Vector Facility; thus, our task was to produce a companion set of scalar versions of all subroutines which produced exactly the same results as the vector versions. Our new set of scalar elementary-function subroutines yield higher speed and

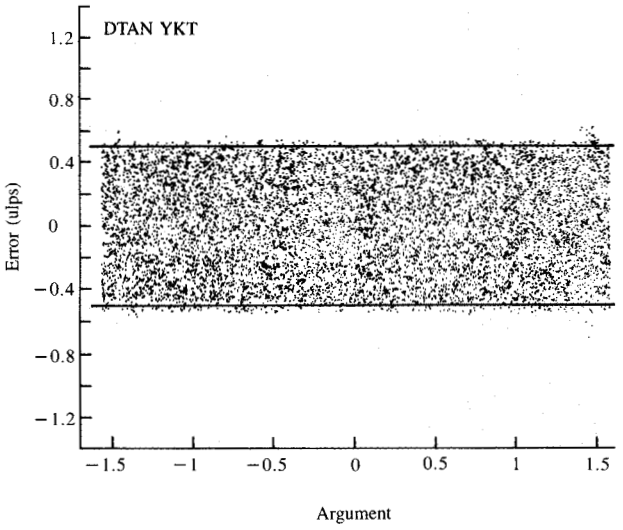

Figure 1

Yorktown DTAN. Plot of errors in the Yorktown DTAN subroutine in ulps for evenly distributed random arguments in the interval $(-\pi/2, \pi/2)$.

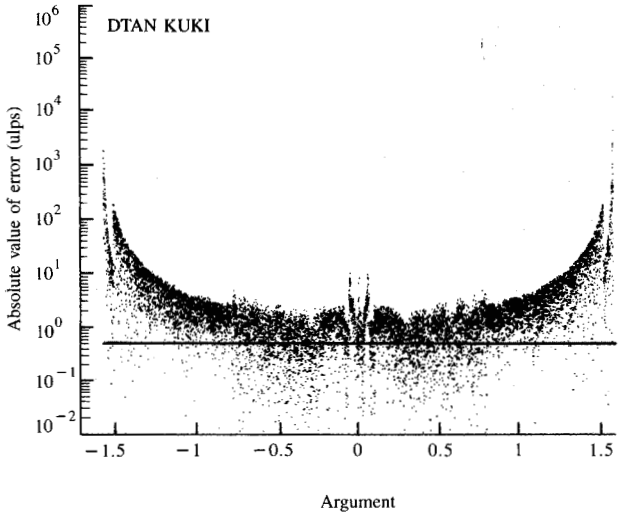

Figure 2

FORTRAN (Kuki) DTAN. Plot of errors in the FORTRAN DTAN subroutine in ulps for evenly distributed random arguments in the interval $(-\pi/2, \pi/2)$. Note that errors are given on a logarithmic scale due to their large size.

greater accuracy in a wide range of System/370 machines. In producing this new library, the new technology afforded us a number of great advantages:

- With more memory available, we could make the subroutines longer and use tables.
- Program development tools such as interactive computing and graphics were available.
- Computer speed and availability permitted extensive testing, including, in some cases, testing every possible argument.

• *Software development tools produced*

To achieve the high performance goal set for the project and to obtain the greatest advantages from the facilities available, a number of programs were developed.

The Vector Facility simulator

In order to write and test programs long before the vector machine was available, Tuckerman wrote a functional simulator permitting one to run and trace modest-sized vector programs on the available System/370 machines. The simulator was very easy to use and its tracing facilities were of enormous help in debugging programs. Its use enabled us to complete most of the vector subroutines long before the Model 3090 with its Vector Facility was available.

The ulp plot

Starting with a demonstration by Moler [11] showing how graphical displays of errors in ulps (*units in* or *of the last place*) can reveal important characteristics of error distributions, Agarwal developed such programs for use on our graphics systems. An ulp is the distance between the two nearest floating-point numbers of the actual result. These programs tested elementary-function algorithms and presented a graphical picture of the distribution of errors. They evolved into a very effective interactive tool for analyzing errors and suggesting strategies for the improvement and correction of our algorithms. Samples of output plots from this program are shown in **Figures 1** through **9**, and they are discussed in Sections 2 and 4.

Polynomial approximation

Polynomial approximation subroutines were available from SL-MATH [12], the mathematical subroutine library. These were revised to compute in extended precision in order to give the high accuracy needed for the double-precision routines. Since much experimentation was needed to get last-bit accuracy, the approximation subroutines were incorporated into interactive programs which computed error distributions and statistics. They also produced machine-readable assembly language and/or FORTRAN statements containing the exact hexadecimal representations of the approximation coefficients.

• *Why do we want correctly rounded results?*

A great deal of satisfaction was obtained from the fact that five of the intrinsic functions reported here always deliver correctly rounded results; these are SQRT, DSQRT, CABS, CDABS, and EXP. One important aspect of this is that

correctly rounded results were obtained with surprisingly little sacrifice in performance. A second, which may have more far-reaching consequences, is that the requirement for future compatibility does not compel one to use the same algorithm when a new machine architecture would make a different algorithm more efficient.

• *Intangibles*

Use of the one-ulp criterion facilitates the preservation of many desired mathematical properties of elementary functions such as monotonicity, symmetry, and important identities. Furthermore, when the result is an exactly representable machine value, the one-ulp criterion guarantees that this result will be obtained. In most cases, the very nature of our algorithm design guaranteed correct symmetry properties. Special attention was paid to the examination of table boundaries in order to ensure that monotonicity was not violated. Of course, monotonicity is not violated for the routines that always deliver correctly rounded results. For other subroutines, large numbers of arguments were tested and no violation of monotonicity was found.

Another desirable property is the preservation of the quadrants in computing the inverse trigonometric functions. Surprisingly, this is an example of a situation where the desire to obtain correctly rounded results was in conflict with the preservation of a mathematical property. For example, DATAN2(Y,X) was made to lie within the quadrant indicated by (X, Y). This means that at times we deliberately incorrectly round the result. In one case, if X is a very small negative number and Y is a very large positive number, the exact result is slightly greater than $\pi/2$. The nearest machine number happens to lie in the first quadrant, but we produce the rounded-up value which is in the second quadrant.

• *Performance of the new programs*

Speed

At first glance, it would seem that the use of a vector algorithm would make it necessary to use the same algorithm for a whole vector of arguments, while a scalar program would test each argument and branch to a routine employing the best method for that argument. Therefore, one might expect the scalar-vector compatibility requirement to cause a loss of performance. By applying special strategies, as described in Section 3, we found that we could keep the performance loss of the scalar subroutines minimal.

Accuracy

For our algorithms, the use of "tuned" tables and other techniques to be discussed below always gives errors of less than one ulp. To be more precise, the result is always one of the two machine numbers bordering the infinite-precision result. Extensive testing showed that correctly rounded results are obtained for more than 95% of the arguments. If any computed value is subsequently found to violate the one-ulp criterion, it will be treated as a program error and the program will be corrected. Our programs were, in many cases, subjected to mathematical analysis which ensured accuracy provided that certain conditions were fulfilled. These conditions, in many cases, were tested numerically to ensure that no violations occurred, and so correct results within one ulp are virtually certain.

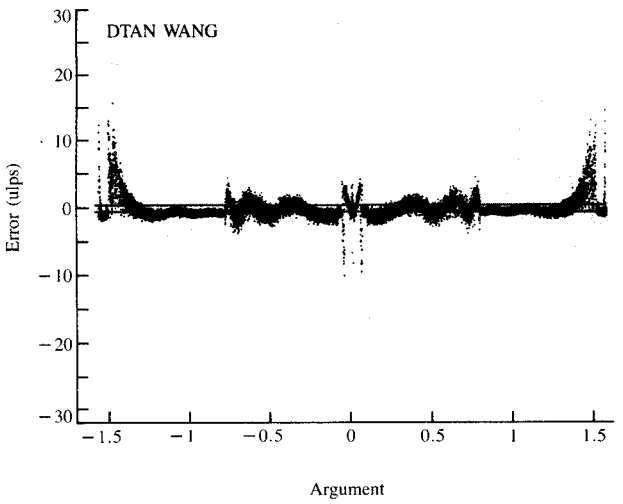

Figure 3

Wang's DTAN. Plot of errors in Wang's DTAN subroutine in ulps for evenly distributed random arguments in the interval $(-\pi/2, \pi/2)$.

Robustness

The old VS FORTRAN library was written with the assumption that little or no effort should be spent on arguments which are unnormalized or subnormal (below the smallest number which can be represented in normalized floating-point format) or which, in some way, may be considered pathological. The present programs give results for both subnormal and unnormalized arguments, where possible.

For arguments near singularities of the tangent function, the old intrinsic TAN function often gave error returns where, in fact, the correct results were machine-representable numbers. It was shown for the present work that for all our intrinsic functions except COTAN near zero, there can be no machine-representable argument so close to a singularity that the result is not a machine-representable number. The present scalar subroutines were written so as to give results correct to within one ulp for all arguments for which normalized machine-representable results exist.

2. Accuracy—The ulp concept

The accuracy of these new programs is described here in terms of units in the last place, or "ulps," and is shown in Figs. 1, 4, and 7 and **Table 1**.

Table 1 Version 2 accuracy statistics (for 10 000 trials): percent correctly rounded, average error, 99th percentile error bound.

FUNCTION DISTRIBUTION (RANGE)	P	EAVE	E99%
ACOS CIRC(0,PI)	99.07	0.25	0.50
DACOS CIRC(0,PI)	97.46	0.25	0.52
ASIN CIRC(-PI/2,PI/2)	99.36	0.25	0.50
DASIN CIRC(-PI/2,PI/2)	95.65	0.25	0.55
ATAN TAN(-PI/2,PI/2)	96.41	0.25	0.54
DATAN TAN(-PI/2,PI/2)	97.58	0.25	0.52
ATAN2 POLAR(16**-16,16**16)	98.23	0.27	0.53
DATAN2 POLAR(16**-16,16**16)	98.79	0.25	0.50
COS LINEAR(-PI,PI)	98.29	0.25	0.51
COS LOG(2**-18*PI,2**18*PI)	98.96	0.23	0.50
DCOS LINEAR(-PI,PI)	96.43	0.25	0.53
DCOS LOG(2**-50*PI,2**50*PI)	96.91	0.20	0.53
SIN LINEAR(-PI,PI)	98.06	0.25	0.51
SIN LOG(2**-18*PI,2**18*PI)	98.74	0.22	0.50
DSIN LINEAR(-PI,PI)	96.62	0.25	0.53
DSIN LOG(2**-50*PI,2**50*PI)	97.60	0.20	0.52
TAN LINEAR(-PI/2,PI/2)	96.89	0.25	0.55
TAN LOG(2**-18*PI,2**18*PI)	97.81	0.22	0.53
DTAN LINEAR(-PI/2,PI/2)	96.39	0.25	0.53
DTAN LOG(2**-39*PI,2**39*PI)	97.56	0.22	0.52
COTAN LINEAR(-PI/2,PI/2)	96.99	0.25	0.54
COTAN LOG(2**-18*PI,2**18*PI)	97.79	0.25	0.53
DCOTAN LINEAR(-PI/2,PI/2)	96.29	0.25	0.53
DCOTAN LOG(2**-39*PI,2**39*PI)	96.31	0.25	0.53
EXP LINEAR(-100,100)	100.00	0.25	0.49
EXP LINEAR(-16,16)	100.00	0.25	0.50
DEXP LINEAR(-100,100)	99.86	0.25	0.49
DEXP LINEAR(-16,16)	99.85	0.25	0.50
ALOG LOG(16**-65,16**63)	100.00	0.25	0.50
DLOG LOG(16**-65,16**63)	96.72	0.25	0.52
ALOG10 LOG(16**-65,16**63)	100.00	0.25	0.50
DLOG10 LOG(16**-65,16**63)	97.07	0.25	0.52
SQRT LOG(16**-65,16**63)	100.00	0.25	0.50
DSQRT LOG(16**-65,16**63)	100.00	0.25	0.50
X**Y LINEAR(.1,10)**60.1	99.99	0.25	0.50
X**Y LOG(16**-65,16**63)**.7	99.99	0.25	0.49
DX**Y LINEAR(.1,10)**60.1	96.66	0.25	0.53
DX**Y LOG(16**-65,16**63)**.7	96.58	0.25	0.53
CABS POLAR(16**-16,16**16)	100.00	0.25	0.49
CDABS POLAR(16**-16,16**16)	100.00	0.25	0.49

• *Floating-point number systems*

We assume that the computer arithmetic is being carried out in a given floating-point number system. Let b be the base, k be the number of base-b digits in the fraction, and l, u be the lower and upper limits of the exponent e of b. Then a floating-point number X can be represented by s, e, a_1, $\cdots$, a_k, where $s = \pm$, $l \le e \le u$, $0 \le a_i < b$, and the associated value is

$$X = s \cdot m \cdot b^e,$$

where

$$0 \le m = \sum_{i=1}^{k} a_i b^{-i} < 1.$$

A nonzero floating-point number is *normalized* if the leading digit a_1 of the fraction is nonzero. The range of positive normalized floating-point numbers X is $b^{l-1} \le X < b^u$.

In the IBM System/370 series of computers, $b = 16$; $k = 6$, 14, or 28 for the short-, long-, or extended-precision formats; and $l = -64$, $u = +63$.

• *The ulp concept*

Consider a positive normalized floating-point number X in such a system. Then a unit in the last place of X is defined as the difference between X and the next larger floating-point number (or between X and b^u if X is the largest floating-point number). In the System/370 representation, if

$$X = m \cdot 16^e,$$

where

$$1/16 \le m < 1,$$

then an ulp of X in that system is

$$\text{ulp}\ (X) = .0 \cdots 01 \cdot 16^e = .10 \cdots 0 \cdot 16^{e-k+1}.$$

For example, an ulp of $.765432 \cdot 16^1$ is $.000001 \cdot 16^1 = .100000 \cdot 16^{-4}$. The digits of the fraction are hexadecimal digits (0, 1, $\cdots$, 9, A, B, $\cdots$, F).

If x is a positive *real* number (of infinite precision) lying in the range of floating-point numbers, then an ulp of x, from the standpoint of the floating-point number system, is

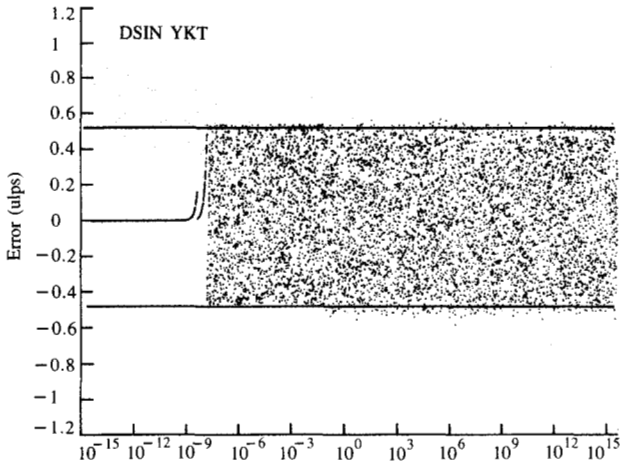

Figure 4

Yorktown DSIN. Plot of errors in the Yorktown DSIN subroutine in ulps for evenly log-distributed random arguments in the interval $(2^{-50}\pi, 2^{50}\pi)$.

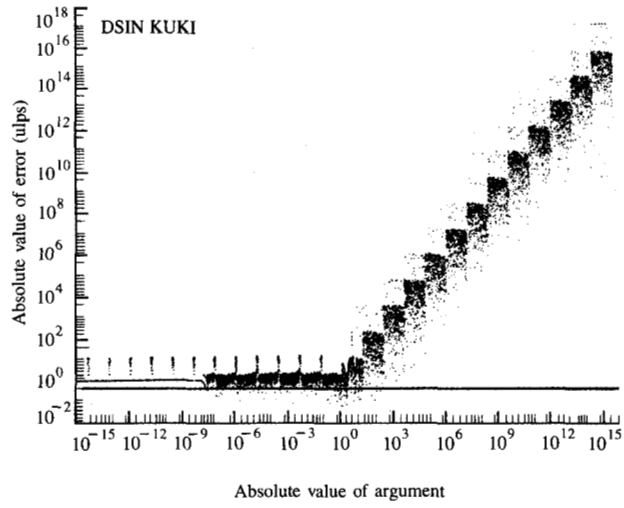

Figure 5

FORTRAN (Kuki) DSIN. Plot of errors in the FORTRAN DSIN subroutine in ulps for evenly log-distributed random arguments in the interval $(2^{-50}\pi, 2^{50}\pi)$. Note that errors are given on a logarithmic scale due to their large size.

defined as an ulp of the largest floating-point number X which does not exceed x, i.e., where X is derived from x by "chopping."

It is convenient to define an ulp of a negative floating-point or real number as the negative of an ulp of its absolute value. An ulp of zero is undefined.

If we are given a computer program which defines a floating-point-valued function $Y = F(X)$ of a floating-point-valued argument X, and which is intended to approximate a given mathematical function $y = f(x)$ (which cannot in general be realized exactly) at floating-point arguments $x = X$, then the absolute (i.e., not relative) signed error in F at a given floating-point argument X is defined as error $(F, f, X) = F(X) - f(X) = Y - y$ = computed value minus true value. It is convenient to express this error in terms of ulps, i.e.,

$$\text{ulp error } (F, f, X) = \text{error } (F, f, X)/\text{ulp } (Y, y).$$

Here ulp (Y, y) is defined as the common value of ulp (Y) and ulp (y), in the usual case when these values are equal. However, in the rare cases when they are unequal, which is when Y and y have different exponents, then ulp (Y, y) is defined as the one of them which has the smaller absolute value. For example, if $y = .100000 \cdot 16^{1} + \varepsilon \cdot \text{ulp } (y)$, where $0 \le \varepsilon < 1/2$, then ulp $(y) = 16^{-5}$; and if $Y = .FFFFFC$, then ulp $(Y) = 16^{-6}$. Then ulp$(Y, y) = 16^{-6}$, and ulp error $(F, f, X) = 16\varepsilon + 4$, correctly indicating a poor approximation, rather than $\varepsilon + 4/16$, which would erroneously indicate a good approximation. A good approximation, of course, is $Y = .100000 \cdot 16^{1} = 1$, for which ulp $(Y, y) = 16^{-5}$, ulp error $(F, f, X) = \varepsilon$. For brevity we write this quantity as e/u.

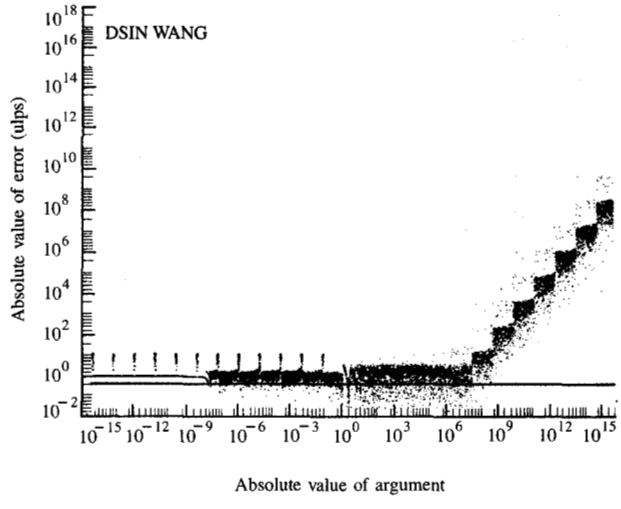

Figure 6

Wang's DSIN. Plot of errors in Wang's DSIN subroutine in ulps for evenly log-distributed random arguments in the interval $(2^{-50}\pi, 2^{50}\pi)$. Note that errors are given on a logarithmic scale due to their large size.

In the new programs, an attempt has been made to minimize the maximum ulp errors. The results are summarized below and in the section on accuracy statistics.

Our SQRT, DSQRT, and EXP functions satisfy $|e/u| < 0.5$; they have best-possible rounding. This is sometimes called the "half-ulp" or "one-point" criterion.

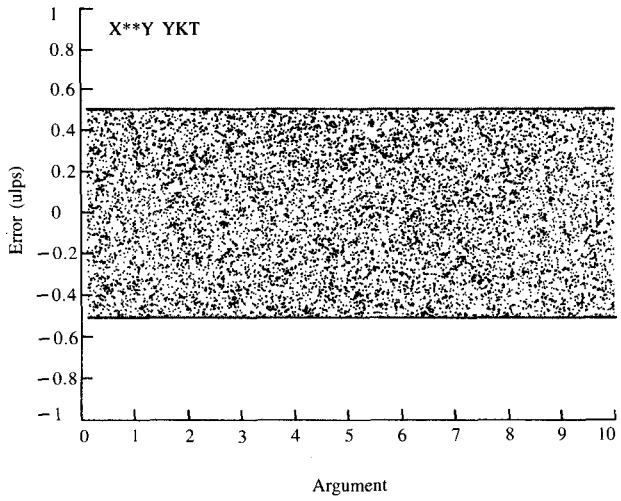

Figure 7

Yorktown X**Y. Plot of errors in the Yorktown single-precision implicit X**Y subprogram in ulps for an evenly distributed random argument X in the interval (.01, 10.) and $Y = 60.1$.

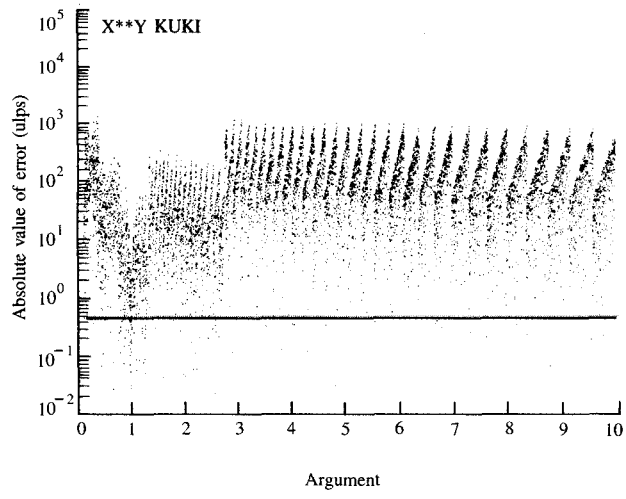

Figure 8

FORTRAN (Kuki) X**Y. Plot of errors in the FORTRAN single-precision implicit X**Y subprogram in ulps for an evenly distributed random argument X in the interval (.01, 10.) and $Y = 60.1$. Note that errors are given on a logarithmic scale due to their large size.

Our CABS and CDABS functions satisfy $|e/u| \leq 0.5$ (this can also be called a half-ulp criterion). They have best-possible rounding, except that unavoidably there are cases when $|e/u| = 0.5$, in which case it would be equally correct to round downward or upward; we choose to round upward. This is consistent with the System/370 definition of rounding. An example in short precision is the following. Let $N = .4 \cdot 16.^6 - 8. = .3FFFF8 \cdot 16^6$, and let $W = X + iY = 3N + 4Ni = .BFFFE8 \cdot 16^6 + .FFFFE0 \cdot 16^6 i$. Then $z = \text{abs}(W) = 5N = .13FFFD8 \cdot 16^7$ lies exactly midway between $.13FFFD \cdot 16^7$ and $.13FFFE \cdot 16^7$. Either of those numbers can be regarded as a rounding Z of the true value, and in either case $|e/u| = 0.5$. One of them (the larger) is returned by CABS, and similarly for CDABS.

This rounding ambiguity can also occur for x^y; for an example in short precision, $258.^3 = (.102000 \cdot 16^3)^3 = .1060C08 \cdot 16^7$, which lies exactly midway between $.1060C0 \cdot 16^7$ and $.1060C1 \cdot 16^7$. However, correct rounding is not always achieved for x^y. The rounding ambiguity cannot occur for any other of our functions. (The only rational solutions x, y of $\log_{10} x = y$ are for y a nonnegative integer. The only algebraic solutions x, y of our remaining $f(x) = y$ are for $x = 0$ in $\sin x$, $\cos x$, $\tan x$, e^x, and for $y = 0$ in their inverse functions [13]. Except for EXP, correct rounding is not always achieved for our other functions.

All of our functions are believed to satisfy $|e/u| < 1.0$; equivalently, a computed function value Y, if not exactly equal to the true value y, is one of the floating-point numbers just above or just below y. We have called this the "one-ulp" or "two-point" criterion. An effort has been made to make the errors $|e/u|$ substantially less than 1.0, and the results can be judged from Table 1 and the ulp plots (Figs. 1, 4, and 7). Note that errors in $|e/u|$ of up to 0.5 are an unavoidable result of any rounding, and we have endeavored to keep the actual errors as nearly within this bound as is practical.

Additionally, the following desirable properties are attained:

1. Special case values which should be exact floating-point numbers are so in fact, e.g., EXP(0.) = 1., SQRT(.25) = .5, $16.^{.25} = 2$. (This is a consequence of the one-ulp criterion.)
2. The $F(X)$ are strictly even or odd functions, i.e., $F(-X) = F(X)$ or $F(-X) = -F(X)$, respectively, for every floating-point number X, if the underlying function $f(x)$ is even or odd, like cosine or sine, respectively. Because of the previous definition of an ulp of a negative number as the negative of an ulp of its absolute value, ulp error (F, f, X) = ulp error $(F, f, -X)$ whenever f and F are either even or odd. (That definition also makes the interpretation of ulp plots of oscillatory functions easier than if an ulp were always considered to be positive, in which case there would be a near-mirroring of the regions $Y \geq 0$ and $Y \leq 0$ onto each other.)
3. The functions are believed to be monotonic where appropriate, although this is not guaranteed.

- *Ulp plots*

The "ulp plots" shown later in this paper are examples of the output of a very useful tool which we developed on hearing from Paul [14] of similar outputs produced by Moler [11].

A specified set of arguments, chosen at random or

linearly, based on a given distribution (e.g., linear or logarithmic) over a given range, is supplied to two functions, typically one which is to be tested and another which is to be used as a reference (e.g., an extended-precision version). The differences are computed in terms of ulps; they can be plotted immediately on a high-resolution graphics terminal and can also be plotted immediately on hard copy if desired. The horizontal axis shows the range of arguments; the vertical axis shows the error in ulps, plotted linearly if its range is small, logarithmically if its range is large.

The ulp plots of our functions are rather featureless, showing a random scattering of points, mostly within ±0.5 ulp, because of the accuracy attained. But during development the ulp plots were very revealing of specific numerical difficulties, much more so than mere statistical summaries would have been. Some of these revelations can be seen in the plots of functions from the older libraries.

For example, the plot (Fig. 5) of DSIN from the VS FORTRAN library versus QSIN shows, for one thing, the loss of ulp precision for large arguments, starting not far above $x = 1$, due to imperfect argument reduction. It also shows, surprisingly, "spikes" of errors of up to 12 ulps (see also Figs. 2, 3, and 6), even for very small arguments [15], where the approximation $\sin x \approx x$ should be accurate to well under 0.5 ulp. These errors resulted from a multiplication of all arguments by $4/\pi$ during range reduction, followed eventually by a polynomial evaluation which in effect multiplied the reduced arguments by $\pi/4$. Whenever the fraction of a floating-point argument lies between $\pi/4$ and 1, the first multiplication yields a fraction between 1 and $4/\pi$ and an exponent increased by 1. This results in a "chopping" loss of nearly a digit of precision, which is, of course, not restored by the second multiplication. Both of these sources of errors have been eliminated in our programs.

The ulp plots and statistics of our functions were made for 10000 or more random arguments, and show no errors as large as one ulp. The value of 10000 was chosen because it nicely exhibits the salient features of our functions. In testing our functions, we ran sample sizes well into the tens of millions. The single-precision functions of one argument were nearly exhaustively tested. This is how we know that the short EXP algorithm delivers the correctly rounded result for all arguments. However, it was not feasible to test a function for all long-precision arguments, nor for all pairs of short-precision arguments (including complex arguments). Our belief that it is possible to produce a library satisfying the one-ulp criterion throughout has been buttressed by analysis, by extra-dense ulp plots in some narrow critical regions such as across boundaries of table intervals and hexadecimal exponent boundaries, and by extensive numerical testing, but it has not been proven in all cases. Any observed violations will be regarded as program errors and will be corrected.

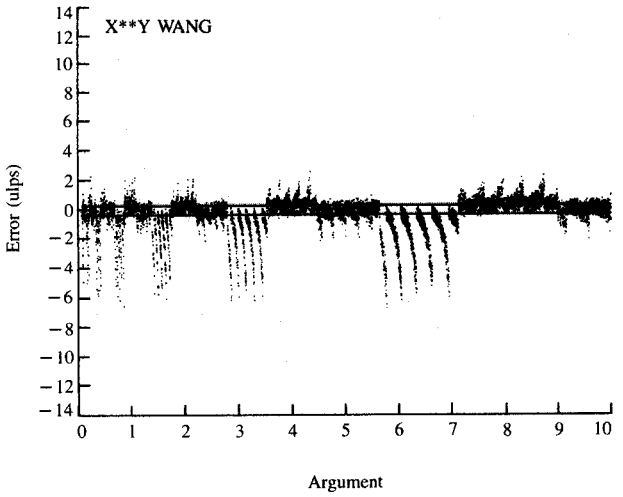

Figure 9

Wang's X**Y. Plot of errors in Wang's single-precision implicit X**Y subprogram in ulps for an evenly distributed random argument X in the interval (.01, 10.) and $Y = 60.1$.

The collection of accuracy statistics for each of the 28 functions, for samples of size 10000 over suitable distributions of arguments, is described in the subsection "Near-correct accuracy." Table 1 shows the number of correctly rounded arguments and the maximum ulp errors observed. **Tables 2** and **3** show the corresponding statistics for the VFORTLIB and VALTLIB (Wang) routines. These tables are described in detail in Section 4.

3. Programming strategy

• *Introduction*

In this section we describe the main concepts and programming methodology that constitute the theory and implementation of our new set of elementary function programs.

Our original task was to produce bitwise-compatible algorithms in both scalar and vector for the Model 3090; that is, the vector and scalar versions should produce identical results for all legal arguments. The divide instruction on the 3090 is a relatively slow instruction compared to multiply, add, and subtract instructions for the vector and scalar. Therefore, almost all of our algorithms avoid divide instructions. Previous scalar routines have not handled unnormalized arguments acceptably. We decided to handle unnormalized arguments in scalar in order to make our functions robust. In the vector hardware, multiplication by an unnormalized nonzero argument produces an unnormalized-operand exception. The scalar hardware does not have this exception. Therefore, we decided not to handle unnormalized nonzero arguments in the vector elementary functions. These arguments may produce unpredictable

Table 2 Version 1 accuracy statistics: percent correctly rounded, average error, 99th percentile error bound.

FUNCTION	DISTRIBUTION (RANGE)	P	EAVE	E99%	TRIALS
ACOS	CIRC(0,PI)	63.34	0.44	1.26	10000
DACOS	CIRC(0,PI)	42.26	0.63	1.47	10000
ASIN	CIRC(-PI/2,PI/2)	68.97	0.38	0.98	10000
DASIN	CIRC(-PI/2,PI/2)	56.68	0.46	1.13	10000
ATAN	TAN(-PI/2,PI/2)	41.74	0.88	3.78	10000
DATAN	TAN(-PI/2,PI/2)	38.82	0.69	2.03	10000
ATAN2	POLAR(16**-16,16**16)	46.27	0.99	8.38	10000
DATAN2	POLAR(16**-16,16**16)	45.29	0.77	4.55	10000
COS	LINEAR(-PI,PI)	35.99	0.74	1.78	10000
COS	LOG(2**-18*PI,2**18*PI)	46.10	0.60	1.66	10000
DCOS	LINEAR(-PI,PI)	12.86	10.93	146.39	10000
DCOS	LOG(2**-50*PI,2**50*PI)	33.87	.79E+14	.17E+16	10000
SIN	LINEAR(-PI,PI)	36.17	0.75	1.83	10000
SIN	LOG(2**-18*PI,2**18*PI)	27.08	1.04	10.19	10000
DSIN	LINEAR(-PI,PI)	15.91	9.76	109.42	10000
DSIN	LOG(2**-50*PI,2**50*PI)	3.87	.79E+14	.17E+16	10000
TAN	LINEAR(-PI/2,PI/2)	37.64	0.84	5.33	10000
TAN	LOG(2**-18*PI,2**18*PI)	35.37	0.98	7.81	9696
DTAN	LINEAR(-PI/2,PI/2)	17.47	18.37	129.23	10000
DTAN	LOG(2**-39*PI,2**39*PI)	14.67	.24E+12	.32E+13	10000
COTAN	LINEAR(-PI/2,PI/2)	38.82	0.84	5.79	10000
COTAN	LOG(2**-18*PI,2**18*PI)	34.93	1.04	6.81	9709
DCOTAN	LINEAR(-PI/2,PI/2)	19.65	24.25	163.46	10000
DCOTAN	LOG(2**-39*PI,2**39*PI)	10.53	.30E+12	.33E+13	10000
EXP	LINEAR(-100,100)	98.43	0.25	0.51	10000
EXP	LINEAR(-16,16)	97.86	0.25	0.52	10000
DEXP	LINEAR(-100,100)	63.76	0.41	1.10	10000
DEXP	LINEAR(-16,16)	64.27	0.40	1.05	10000
ALOG	LOG(16**-65,16**63)	74.94	0.34	0.99	10000
DLOG	LOG(16**-65,16**63)	57.06	0.50	1.47	10000
ALOG10	LOG(16**-65,16**63)	67.80	0.57	4.75	10000
DLOG10	LOG(16**-65,16**63)	58.81	0.85	8.05	10000
SQRT	LOG(16**-65,16**63)	97.21	0.25	0.54	10000
DSQRT	LOG(16**-65,16**63)	100.00	0.25	0.50	10000
X**Y	LINEAR(.1,10)**60.1	1.29	124.57	691.65	10000
X**Y	LOG(16**-65,16**63)**.7	3.35	37.14	194.40	10000
DX**Y	LINEAR(.1,10)**60.1	0.61	154.86	874.08	10000
DX**Y	LOG(16**-65,16**63)**.7	2.20	53.99	282.56	10000
CABS	POLAR(16**-16,16**16)	37.41	0.77	2.12	10000
CDABS	POLAR(16**-16,16**16)	38.18	0.78	2.13	10000

results. All arguments with zero fraction are correctly handled, regardless of whether the exponent is zero. Thus we claim bitwise compatibility between the new scalar and vector routines for all arguments except nonzero unnormalized operands.

Under this requirement we wanted our functions to be as accurate as and to execute faster than the current VS FORTRAN product. The very stringent requirement of bitwise compatibility restricted the speed of both the vector and scalar algorithms. Our requirement was to weigh scalar and vector algorithms equally in our attempt to meet the above goals. Some remarks on the way we handled vector/scalar trade-offs now follow. In our vector codes, only those arguments are handled which would have been processed in the main path(s) of the scalar code. All arguments which require special processing are handled by the scalar code, either by branching to the appropriate scalar function or by duplicating some of the scalar code as part of the vector function. For trigonometric functions, more than 90% of the arguments between zero and $\pi/2$ are handled in the vector mode. For other functions, almost all of the arguments in a certain distribution are handled in the vector mode. Some of the elementary-function routines always produce correctly rounded results. For these routines, different algorithms can be used in the scalar and the vector mode, while preserving the bitwise compatibility feature of the library. This fact was used to redesign the vector algorithms for some of the correctly rounded vector functions so that they could be independently optimized for better performance on the vector hardware.

In this section, we cover the following topics: Tuckerman rounding (see [16(a), p.10] and [16(b), p. 14]), table-based approach, near-correct accuracy, fast-track programming, robustness, and error handling. Tuckerman rounding is a simple multiplicative algebraic/numeric condition that allows us to produce correctly rounded results for the square root and CABS functions. Hull [17] has also described a condition, but his condition requires higher-precision arithmetic. Kahan [18] has also discovered a similar condition; his result uses a divide instruction. Tuckerman's

Table 3 VALTLIB accuracy statistics (for 10 000 trials): percent correctly rounded, average error, 99th percentile error bound.

FUNCTION DISTRIBUTION (RANGE)	P	EAVE	E99%
DCOS LINEAR(-PI,PI)	21.51	1.40	4.16
DCOS LOG(2**-50*PI,2**50*PI)	41.88	.46E+13	.16E+09
DSIN LINEAR(-PI,PI)	24.66	1.36	4.16
DSIN LOG(2**-50*PI,2**50*PI)	8.57	.73E+07	.15E+09
DTAN LINEAR(-PI/2,PI/2)	36.68	1.02	7.57
DTAN LOG(2**-39*PI,2**39*PI)	24.21	.16E+05	.17E+06
DCOTAN LINEAR(-PI/2,PI/2)	30.25	1.01	4.87
DCOTAN LOG(2**-39*PI,2**39*PI)	19.43	.17E+05	.15E+06
EXP LINEAR(-100,100)	98.43	0.25	0.51
EXP LINEAR(-16,16)	97.86	0.25	0.52
DEXP LINEAR(-100,100)	63.76	0.41	1.10
DEXP LINEAR(-16,16)	64.27	0.40	1.05
X**Y LINEAR(.1,10)**60.1	56.45	0.81	5.01
X**Y LOG(16**-65,16**63)**.7	98.94	0.25	0.50
DX**Y LINEAR(.1,10)**60.1	27.27	1.45	6.82
DX**Y LOG(16**-65,16**63)**.7	67.48	0.39	1.32

condition is of historic significance, as its use allowed us to produce IBM's first elementary function that delivered correctly rounded results for all arguments.

The table-based approach follows naturally from a fundamental idea: Let an elementary function be expressed as the sum of some exact value and a small correction, cf. Kuki and Ascoly [6], Fullerton [19]. We developed several ways to find such an exact value. For example, we adopted a strategy of sharing one large (about 256 entries) table over 14 routines (TAN, COT, ATAN, ATAN2) in both short and long, vector and scalar. (We did not vectorize short-precision TAN and COT.) We originally coded SQRT and CABS using this technique (eight routines sharing a table of size 192 entries of 12 bytes per entry). In our TAN/COT/ATAN/ATAN2 scalar/vector programs, we use a common table where, for each table entry, an x_0 is chosen near the middle of the table interval such that tan (x_0) is an exact short word and x_0 is stored as a double word followed by its short-word continuation (hexadecimal digits 15–20). Actually, x_0 is a transcendental number which we approximate to 20 hex digits. In our approach, with the same amount of table storage (16 bytes per table entry) we are able to get up to 20-digit precision. The only disadvantage of this approach over the next one is that it requires an extra addition (to account for the continuation of x_0).

We heard about another way, called the Accurate Table Method, from the work of Gal [20]. Kahan has also informed us that Miller [21] wrote about the extra-accurate table idea in 1958. Gal's idea helps in eliminating the need to store the continuation (beyond the 14th digit) of the function value. Typically, the last digits of x_0 (a double word) are chosen so that typically hex digits 15–17 of $f(x_0)$ are zero. Thus, with 16 bytes of storage per table entry, approximately 17-digit precision can be obtained. This method saves one floating-point addition over the alternative approach above that develops an exact value. There is another way to save arithmetic; for example, the table for DEXP is an accurate table with a constant built into it so that an extra floating-point add can be saved. For other codes (DSIN/DCOS, DASIN/DACOS) we developed a new concept of built-in rounding. Combining the use of this type of "accurate table" with Gal's method saves an extra addition.

We usually get correctly rounded results by keeping the correction term sufficiently small. When the correction term is then added to the exact values a right shift occurs, usually producing the correctly rounded result. We chose to make this right shift at least two hexadecimal digits. This choice produced correctly rounded results more than 95% of the time.

The idea behind fast-track programming is to judiciously choose a series of cheap tests which filter out difficult arguments that rarely occur. These rare arguments require extra computing; our approach is to do more computing only when necessary. The normal arguments filter quickly into the main path of the code where the instructions are few so that the execution is fast.

Robustness and error handling are features that cost very little. Error handling is a rare event; we presume that it will not occur. Only when an error-producing argument is detected by the code do we set up the complete subroutine linkage necessary for a trace-back. Hence, most of the time we pay only the cost of a minimal subroutine linkage. In the text to follow, we discuss error handling as part of fast-track programming.

The term robustness refers mainly to delivering correctly rounded results for unnormalized arguments [22] and to being precise at overflow and underflow boundaries. These problems rarely occur in the FORTRAN environment. Again, we have cheap or no-cost tests to detect unnormalized operands. The benefit to the user is that he knows he will never get garbage from the scalar elementary-function routines; for all inputs in the domain of the elementary function the result is correct within one ulp.

• *Tuckerman rounding*

The previous square-root routines, SQRT and DSQRT, usually return the correctly rounded square root, but not always. The new routines always do, by the following means, which we call Tuckerman rounding.

If x, $\bar{y}$, y are positive floating-point numbers, such that $\bar{y}$ is the (possibly unknown) nearest floating-point number to $\sqrt{x}$, and y is a "candidate" to be $\bar{y}$, produced by some approximation, then $y = \bar{y}$ if and only if

$$(y_- + y)/2 < \sqrt{x} < (y + y_+)/2$$

(equalities cannot occur), where y_- and y_+ are the floating-point numbers just below and just above y. The terms in these inequalities cannot be evaluated directly on the computer, since $\sqrt{x}$ is irrational and unknown, and the other terms of the inequality are not floating-point numbers of the given precision. However, these inequalities can be shown to be equivalent to

$$y_- * y < x \leq y * y_+ ,$$

where $*$ denotes System/360/370 multiplication (which truncates the result), so that the tests are easily carried out without the need for extra precision. (Note the asymmetry: one $<$, one $\leq$.) If the left inequality fails, y is too large; if the right inequality fails, y is too small.

It is convenient to first use an approximation which is known to give y in the range

$$\sqrt{x} - 1.5 \text{ ulp} < y < \sqrt{x} + 0.5 \text{ ulp}.$$

Then it is sufficient to test whether

$$x \leq y * y_+ ,$$

where $y_+ = y + \text{ulp}(y)$. If the inequality holds, $\bar{y} = y$; otherwise, $\bar{y} = y_+$.

A modification of these tests is used to achieve correct rounding in CABS and CDABS (absolute value of a complex number). Here, however, in some cases the true function value can lie exactly between two consecutive floating-point numbers; in these cases the larger is chosen.

• *Table-based approach*

A central idea in obtaining high accuracy for our elementary functions is the following: Let

$$ef(x) = EXACT(x_0) + corr(x, x_0), \tag{1}$$

where *ef* stands for an elementary function, *EXACT* is some machine number that represents $ef(x_0)$ exactly or to higher precision, and *corr* is approximated by *CORR*, a small—usually two orders of magnitude smaller—machine number that is computed. The value x_0 is either *not* a machine number or some specially chosen machine number that makes $ef(x_0)$ especially precise, where $x - x_0$ is again about two orders of magnitude smaller than x and x_0. Suppose that

$$|corr| < \frac{1}{256} |EXACT| . \tag{2}$$

Then an ulp of $ef(x)$ is at least 256 times an ulp of *corr*, and several floating-point operations will, in worst case, contribute no more than, say, 5/256-ulp error to the computation of *corr* by *CORR*, a polynomial minimax approximation to *corr*. Now $|CORR - corr| < 5/256$ ulp, and, if desirable, a rounding can be added to *CORR*. The final addition of the table value and the correction term always introduces an error in the range zero to one ulp (if both the terms are of the same sign) or $-1/16$ ulp to 15/16 ulp (if they are of opposite signs) because of the properties of the IBM System/370 floating-point Add and Subtract operations. This is the largest single source of error in our computation. Moreover, this error is biased with an average of 1/2 ulp (if the terms are of the same sign) or 7/16 ulp (if they are of opposite signs). We compensate for this bias by incorporating a compensatory term of 15/32 ulp (which is the average of 1/2 ulp and 7/16 ulp) in *CORR*. This imperfect compensation of bias adds 1/32 ulp to our overall error. Thus the final floating addition, $EXACT + CORR$, is made with absolute error $\leq 1/32$ ulp, and hence $ef(x)$ can be computed with error no more than, say, 13/256 ulp. For our elementary functions these facts yield the correctly rounded $ef(x)$ most of the time, and with error less than one ulp for *all* x. For some functions, such as DEXP, the relative signs of the two terms are fixed, and for these functions we can apply perfect compensation for the bias error. This results in 99.8% correctly rounded results for DEXP.

A consequence of (1), (2), and the fact that $ef(x)$ is differentiable is that the set x_0 of points needed to compute $ef(x)$ for arbitrary x is about size 256. The function *CORR* approximating *corr* is a linear combination of polynomials in the variable $\Delta x = x - x_0$. The constants in the linear combination are functions of x, x_0, and Δx, and these can be easily and cheaply represented in tables. An unexpected benefit of this approach is that *CORR* can be computed cheaply, mainly because the polynomials in Δx have low degree. Thus the table-based approach simultaneously produces a method to compute $ef(x)$ both more accurately and faster than the approach of using a single polynomial or rational approximation.

• *Near-correct accuracy*

All the functions produce results which are strictly less than one ulp away from the infinite-precision results. One implication of this is that if the infinite-precision result is machine-representable (i.e., it is a valid machine number), it is produced by these functions. This is particularly important for functions such as square root, absolute value of a complex number, and the power function. For these functions, the result is often an exact machine-representable number, which our routines produce. Most elementary function libraries do not guarantee that. As an example,

X**1.0 will always be X for our functions, while for some other libraries the result could be many ulps away from X, sometimes as much as hundreds of ulps away. Also, X**2.0 usually produces a correctly rounded value, while X*X always produces the truncated value of x^2.

In computing the elementary functions, one can always obtain sufficient accuracy by doing arithmetic in higher precision. On most IBM System/370 machines, except the low-end machines, short-precision operations take about as much time as long-precision operations. Therefore, we decided to take advantage of this to obtain good accuracy for short-precision routines. On all IBM System/370 machines, extended-precision routines are considerably more expensive. On the 3090 Vector Facility, they are not supported. Therefore, we could not rely on extended-precision operations to deliver the desired accuracy for the long-precision routines. The tables were designed to deliver the desired accuracy for the long-precision routines. In most cases, the same tables were used to compute the short-precision functions. Except for EXP, short-precision routines were not independently designed.

Since short-precision routines use mostly long-precision operations, the final rounding was easily and accurately accomplished by the LRER (load-rounded short-precision from long) instruction. This explains why most short-precision routines have close to 100% correctly rounded results. On the negative side, use of long-precision operations slows down short-precision routines on the low-end machines.

• *Fast-track programming and error handling*

One of the standard practices in assembly language programming is to place at the beginning of the program a header or "eye-catcher" identifying the program. This is useful for trace-back in case of an error but slows down the code, as an extra branch is required. For the scalar elementary-function routines, where the total number of instructions executed is small, this overhead becomes significant. Since our routines are robust and do not produce an unexpected error (such as an intermediate underflow or overflow), we have eliminated the header, which speeds up the execution for most of the arguments. In the rare case where the argument or result is out of range, the registers are appropriately modified to point to a correct header, so that proper error messages and error trace-back information can be given to the caller. This technique was used by Scarborough [10] when he sped up the original IBM library designed by Kuki.

In coding the scalar elementary-function routines, a great deal of effort was spent in minimizing the number of general registers used, as they need to be saved and restored, except that by convention the registers R0 and R1 need not be saved and restored. Therefore, R0 can always be used and R1 can be used after it is no longer needed to fetch the arguments or for error traceback. Additional registers, if needed, are saved and restored in a double-word-aligned area. The registers are restored just before the final result is computed. This facilitates use of the conditional branch instructions to exit from the routine. In many situations, a test is made and for certain conditions we exit from the routine, or else we do more processing before exiting from the routine.

Similarly, in coding the vector elementary functions, a great deal of effort was spent in minimizing the number of vector registers used, since they need to be saved and restored if they are in use, as signaled by the calling program.

In coding these functions, we have optimized performance for what we consider a reasonable distribution of the arguments, although all arguments are properly handled. These "most-likely" arguments are handled in the main path of the routine with no branches or minimal branches taken. The unusual arguments which require extra processing or somewhat different processing are quickly filtered out and processed by branching to different segments of the program. The program flow can be described as a tree structure. The very first test in the main path usually routes most of the unusual arguments to some point of control. At that point, further tests are done to properly classify and process the argument. This minimizes the number of tests to be made in the main path of the code, leading to an efficient implementation for the most likely arguments.

As mentioned above, for some arguments extra processing may be required. This extra processing may take the form of some preprocessing and postprocessing, with essentially the main path code in between. In this situation, after the problem argument has been identified and preprocessed, we save R14 (the return address register) and modify it to point to a postprocessing label. Then we branch into the main path, where after all the processing BR 14 automatically branches to the postprocessing label. There, extra processing is carried out, and R14 is restored to its original value for exit from the routine. This procedure eliminates the need for additional tests in the main path of the code.

The vector functions do not handle error reporting for the arguments. The arguments which require error messages are handled by the scalar code. In the vector code, some quick checks are made for the possibility of error reporting, and in that case, either all the arguments or some of the arguments are processed by calling the corresponding scalar function.

The vector-mask-mode instructions are heavily used to do conditional operations, which in the scalar code are normally handled by branching. By doing a few mask-mode operations we are able to combine many main paths of the scalar code into one single vector code. This helps in maintaining close to full vector lengths for all the operations. Where appropriate, vector-store-compressed (VSTK/VSTKE/VSTKD) instructions are used to separate the arguments into two or more cases or to collect the

exceptional arguments. The separated arguments are processed in either the scalar or the vector mode depending on the specific situation, and the results are inserted into the result vector by vector-load-expanded (VLY/VLYE/VLYD) instructions.

Care is taken not to produce any intermediate underflows. This problem can sometimes be avoided by masking out the underflow mask bit in the program status word (PSW). For the scalar code, where only one argument is processed in each call, the extra cycles required to mask out the underflow bit and to restore it on exit are not justified, and so other coding precautions are taken. But in the vector code, where many arguments are processed in a single call, the extra cost of saving and restoring the underflow bit is justified. This is done for some of the vector functions to improve performance.

- *Robustness*

The new functions handle unnormalized arguments correctly in scalar mode. For unnormalized arguments, a normalized result is produced with an accuracy of one ulp (or 0.5 ulp, in the case of perfect rounding). (For the functions with one-ulp accuracy, in rare cases, this result may be one ulp away from the result produced by the equivalent normalized argument.) The previous library did not always handle unnormalized arguments correctly. Often the results were meaningless.

Another feature of the library is that the argument and result boundaries are strictly observed. For example, for the power function, if the result is within the range of machine-representable numbers (i.e., it does not underflow or overflow), it is always produced without any error messages. In the previous VS FORTRAN library, for a large number of machine-representable results close to underflow/overflow boundaries, an underflow or overflow message was issued and the result was not computed. This was true for many functions. For all the new functions, if the exact result is within the underflow and overflow range of the machine, it is computed with one-ulp accuracy. Furthermore, in the computing process, intermediate underflows and overflows are strictly avoided. This has been achieved by appropriate scaling, if the possibility of an underflow or overflow exists. After the scaled result is computed, it is examined to see if the final result is going to be within the machine-representable range. The previous library produced intermediate underflows for many functions.

4. Performance

- *Speed*

In this section we compare the speed and accuracy of the new elementary functions to those in the present libraries (VFORTLIB and VALTLIB).

Tables 4–7 present information on speed. Table 4 gives the time per call (in microseconds) for the new functions, averaged over 10000 random arguments on the machines indicated. These times were obtained by timing a FORTRAN loop like

```
  DO 1 J = 1,N
1 Z(J) = F(X(J))
```

compiled with VS FORTRAN 1.4.0 (with $N = 10000$) and dividing the resulting time by N. The time listed is the median of three trials (with different random arguments). This procedure means that the times include loop and subroutine-linkage overhead. To give the reader an indication of the magnitude of this overhead we have included "dummy" times, which are timings for functions F which consist of an immediate return to the caller (a BR 14). There are six of these, depending on whether F is a single- or double-precision function of one or two real or one complex argument. For some reason, in the DO-loops we used, the FORTRAN compiler treats X(J)**Y(J) differently from other functions of $X(J)$ and $Y(J)$; hence, the two-argument dummy times only apply to (D)ATAN2. Timings for the new functions depend on what distribution of arguments is assumed. We have attempted to choose representative distributions. The times will increase when scaling is required to avoid intermediate underflow or overflow. For the inverse trigonometric functions, we assume that the result is uniformly distributed in the indicated range. The logarithmic distribution means that the arguments are chosen in the indicated interval in such a way that their logarithms are uniformly distributed. The polar distribution means that we choose a pair of arguments $r \cos\theta$, $r \sin\theta$ so that r is logarithmically distributed in the indicated range and θ is uniformly distributed in $(0, 2\pi)$. For the power function, X varies as indicated while Y is held constant. For the trigonometric functions, the logarithmic distribution in effect averages the speed on large arguments (which require precise argument reduction) with the speed on small arguments (which may take a special fast track through the code). So that the interested reader may separate these ranges, we have indicated below each of these times what percentage of the total time is contributed by arguments in the lower and upper halves of the logarithmic range, respectively. This separation is not always possible for vector routines, but in these cases it is roughly correct.

Table 5 lists the ratio of the VFORTLIB times to the new function times (as given in Table 4). Since there are no vector functions in VFORTLIB, Column 7 is the ratio of the times for the scalar routines in VFORTLIB to the times for the new vector routines. Table 6 similarly compares the VALTLIB times to those of the new functions. We include in Table 6 all the functions in VALTLIB.

Table 7 gives three sets of ratios. Column 1 is the ratio of the times for the old library on the 3081KX to the times for the new scalar library on the 3090. Column 3 is the ratio of the times for the old library on the 3081KX to the times for

Table 4 Kernel measurements in microseconds for new scalar elementary functions (VS FORTRAN Version 2) by model.

FUNCTION	DISTRIBUTION (RANGE)	4331-2	4361-5	4341-2	4381-3	3081KX	3090-S	3090-V
EDUM		9.90	3.07	4.88	2.19	.85	.46	
E2DUM		12.21	3.93	5.66	2.61	.96	.55	
CDUM		10.58	3.17	5.18	2.43	.84	.47	
DDUM		10.71	3.54	5.05	2.22	.85	.49	
D2DUM		13.47	4.63	5.72	2.72	1.00	.56	
CDDUM		11.41	3.74	5.12	2.53	.86	.50	
ACOS	CIRC(0,PI)	179.62	33.28	33.23	12.57	4.56	2.06	
DACOS	CIRC(0,PI)	496.24	61.73	64.47	20.01	7.59	3.09	
ASIN	CIRC(-PI/2,PI/2)	174.28	30.72	34.59	13.05	4.50	2.14	
DASIN	CIRC(-PI/2,PI/2)	481.10	56.79	62.68	19.21	7.26	3.01	
ATAN	TAN(-PI/2,PI/2)	102.96	25.62	28.93	12.81	4.00	2.04	1.41
DATAN	TAN(-PI/2,PI/2)	360.42	53.49	51.19	19.70	6.63	2.99	1.68
ATAN2	POLAR(16**-16,16**16)	152.27	39.41	41.78	18.79	5.83	2.94	1.76
DATAN2	POLAR(16**-16,16**16)	463.13	72.55	67.17	26.30	9.23	4.21	2.16
COS	LINEAR(-PI,PI)	102.38	23.33	25.68	9.99	3.51	1.56	.82
COS	LOG(2**-18*PI,2**18*PI)	161.84	39.94	34.13	12.37	4.48	2.07	.82
	SUBRANGE PERCENTAGES	(32,68)	(30,70)	(37,63)	(38,62)	(38,62)	(38,62)	(46,54)
DCOS	LINEAR(-PI,PI)	346.94	42.49	46.32	14.77	5.83	2.23	1.15
DCOS	LOG(2**-50*PI,2**50*PI)	377.66	53.85	49.98	16.18	6.20	2.53	2.02
	SUBRANGE PERCENTAGES	(29,71)	(28,72)	(31,69)	(32,68)	(33,67)	(34,66)	(25,75)
SIN	LINEAR(-PI,PI)	105.72	24.13	27.08	12.66	3.61	1.73	.86
SIN	LOG(2**-18*PI,2**18*PI)	150.54	36.88	32.14	11.90	4.17	1.97	1.28
	SUBRANGE PERCENTAGES	(24,76)	(22,78)	(32,68)	(31,69)	(34,66)	(35,65)	(66,34)
DSIN	LINEAR(-PI,PI)	338.78	40.95	46.18	14.26	5.61	2.28	1.09
DSIN	LOG(2**-50*PI,2**50*PI)	372.68	59.26	51.10	16.99	6.26	2.60	2.71
	SUBRANGE PERCENTAGES	(28,72)	(21,79)	(30,70)	(30,70)	(32,68)	(32,68)	(43,57)
TAN	LINEAR(-PI/2,PI/2)	143.20	33.11	34.57	14.05	4.75	2.28	
TAN	LOG(2**-18*PI,2**18*PI)	176.30	32.08	36.64	13.97	4.97	2.27	
	SUBRANGE PERCENTAGES	(28,72)	(33,67)	(36,64)	(38,62)	(37,63)	(38,62)	
DTAN	LINEAR(-PI/2,PI/2)	442.53	65.01	56.57	21.02	7.70	3.30	2.29
DTAN	LOG(2**-39*PI,2**39*PI)	481.85	86.56	62.72	23.68	8.07	3.78	3.39
	SUBRANGE PERCENTAGES	(25,75)	(20,80)	(28,72)	(27,73)	(31,69)	(29,71)	(36,64)
COTAN	LINEAR(-PI/2,PI/2)	142.92	33.18	34.29	14.49	4.72	2.28	
COTAN	LOG(2**-18*PI,2**18*PI)	197.97	38.27	39.51	15.51	5.37	2.55	
	SUBRANGE PERCENTAGES	(36,64)	(44,56)	(40,60)	(45,55)	(42,58)	(44,56)	
DCOTAN	LINEAR(-PI/2,PI/2)	442.98	64.96	56.41	21.36	7.68	3.27	2.28
DCOTAN	LOG(2**-39*PI,2**39*PI)	620.00	104.14	76.84	28.95	9.62	4.49	4.11
	SUBRANGE PERCENTAGES	(42,58)	(33,67)	(41,59)	(40,60)	(43,57)	(40,60)	(47,53)
EXP	LINEAR(-100,100)	227.40	20.35	32.99	8.56	3.51	1.74	.62
EXP	LINEAR(-16,16)	254.82	39.14	37.46	10.25	3.85	1.97	.62
DEXP	LINEAR(-100,100)	445.59	76.55	65.91	21.68	7.30	3.04	.89
DEXP	LINEAR(-16,16)	445.43	76.55	65.73	21.67	7.30	3.03	.88
ALOG	LOG(16**-65,16**63)	232.36	48.86	37.84	13.47	4.40	2.01	.74
DLOG	LOG(16**-65,16**63)	431.27	74.66	63.06	20.88	7.07	3.05	.94
ALOG10	LOG(16**-65,16**63)	284.06	53.23	45.44	15.39	5.28	2.46	.77
DLOG10	LOG(16**-65,16**63)	594.90	105.90	80.92	25.23	9.16	3.56	1.14
SQRT	LOG(16**-65,16**63)	153.86	25.65	34.83	13.18	4.40	2.15	.68
DSQRT	LOG(16**-65,16**63)	279.65	52.75	46.95	19.94	6.46	3.21	.94
X**Y	LINEAR(.1,10)**60	481.57	81.91	73.37	25.44	8.62	3.69	1.26
X**Y	LOG(16**-65,16**63)**.7	480.91	81.93	73.54	25.69	8.63	3.69	1.26
DX**Y	LINEAR(.1,10)**60	1032.14	125.99	128.40	41.89	15.38	5.76	2.18
DX**Y	LOG(16**-65,16**63)**.7	1030.97	125.94	128.72	41.62	15.39	5.75	2.18
CABS	POLAR(16**-16,16**16)	233.07	33.96	46.76	17.40	5.97	2.66	.92
CDABS	POLAR(16**-16,16**16)	639.55	80.07	86.04	31.49	10.78	4.64	1.81

the new vector library on the 3090 (with Vector Facility). For full vector lengths, a user will get this gain when he moves from the 3081KX to the 3090 Vector Facility. Column 2 is the ratio of the times of the new scalar library to the times of the new vector library on the 3090.

Generating precise times is difficult, since seemingly inconsequential changes in the timing procedure may have a noticeable effect on the measured times. For example, on the 3081KX the performance of the STM and LM instructions is severely degraded near page boundaries. This means that in the rare event that the save area of a subroutine is near a page boundary, the speed of execution of the subroutine will be substantially decreased. Also, the functions timed are not necessarily the final versions, as changes are being made. For these reasons the reader should interpret Tables 4–7 as giving a general indication rather than a precise measurement of what performance (in terms of speed) users may expect from the new library.

• *Accuracy*

Tables 1–3 give accuracy figures for the new library, VFORTLIB, and VALTLIB, respectively. For each function

Table 5 Ratios of VS FORTRAN Version 1 library measurements to Version 2 counterparts.

FUNCTION	DISTRIBUTION (RANGE)	4331-2	4361-5	4341-2	4381-3	3081KX	3090-S	3090-V
ACOS	CIRC(0,PI)	1.16	1.86	1.89	2.53	2.13	2.32	
DACOS	CIRC(0,PI)	1.37	2.71	1.64	2.71	2.07	2.57	
ASIN	CIRC(-PI/2,PI/2)	1.20	2.02	1.79	2.39	2.16	2.26	
DASIN	CIRC(-PI/2,PI/2)	1.41	2.94	1.67	2.80	2.17	2.63	
ATAN	TAN(-PI/2,PI/2)	1.54	1.61	1.67	1.67	1.72	1.59	2.30
DATAN	TAN(-PI/2,PI/2)	1.59	2.25	1.59	1.96	1.74	1.87	3.33
ATAN2	POLAR(16**-16,16**16)	1.28	1.32	1.43	1.48	1.54	1.44	2.40
DATAN2	POLAR(16**-16,16**16)	1.42	1.97	1.44	1.78	1.55	1.65	3.22
COS	LINEAR(-PI,PI)	1.63	1.25	1.57	1.43	1.42	1.42	2.71
COS	LOG(2**-18*PI,2**18*PI)	.98	.69	1.10	1.06	1.04	1.01	2.55
DCOS	LINEAR(-PI,PI)	1.55	1.58	1.45	1.51	1.38	1.48	2.86
DCOS	LOG(2**-50*PI,2**50*PI)	1.44	1.29	1.27	1.38	1.25	1.28	1.61
SIN	LINEAR(-PI,PI)	1.58	1.20	1.49	1.11	1.37	1.27	2.55
SIN	LOG(2**-18*PI,2**18*PI)	1.10	.76	1.21	1.14	1.14	1.08	1.66
DSIN	LINEAR(-PI,PI)	1.59	1.64	1.46	1.57	1.44	1.44	3.02
DSIN	LOG(2**-50*PI,2**50*PI)	1.44	1.15	1.24	1.28	1.22	1.23	1.18
TAN	LINEAR(-PI/2,PI/2)	1.41	1.23	1.46	1.44	1.48	1.43	
TAN	LOG(2**-18*PI,2**18*PI)	1.28	1.44	1.56	1.66	1.63	1.72	
DTAN	LINEAR(-PI/2,PI/2)	1.28	1.30	1.36	1.42	1.33	1.38	1.99
DTAN	LOG(2**-39*PI,2**39*PI)	1.17	.97	1.23	1.28	1.26	1.20	1.34
COTAN	LINEAR(-PI/2,PI/2)	1.44	1.25	1.52	1.46	1.54	1.47	
COTAN	LOG(2**-18*PI,2**18*PI)	1.17	1.25	1.50	1.56	1.57	1.60	
DCOTAN	LINEAR(-PI/2,PI/2)	1.29	1.32	1.39	1.46	1.36	1.42	2.03
DCOTAN	LOG(2**-39*PI,2**39*PI)	.92	.83	1.03	1.09	1.09	1.05	1.14
EXP	LINEAR(-100,100)	.75	1.81	1.32	2.42	1.90	1.84	5.18
EXP	LINEAR(-16,16)	.65	.92	1.15	1.99	1.70	1.61	5.11
DEXP	LINEAR(-100,100)	1.18	1.08	1.16	1.12	1.19	1.13	3.85
DEXP	LINEAR(-16,16)	1.18	1.07	1.16	1.10	1.17	1.12	3.84
ALOG	LOG(16**-65,16**63)	.74	.91	1.24	1.44	1.43	1.49	4.04
DLOG	LOG(16**-65,16**63)	1.35	1.57	1.18	1.45	1.37	1.46	4.73
ALOG10	LOG(16**-65,16**63)	.65	.86	1.11	1.33	1.24	1.24	3.96
DLOG10	LOG(16**-65,16**63)	1.05	1.13	.97	1.21	1.10	1.28	4.00
SQRT	LOG(16**-65,16**63)	.70	1.34	.96	1.42	1.22	1.28	4.04
DSQRT	LOG(16**-65,16**63)	.96	1.46	1.08	1.38	1.21	1.27	4.33
X**Y	LINEAR(.1,10)**60	.72	.96	1.20	1.53	1.44	1.56	4.58
X**Y	LOG(16**-65,16**63)**.7	.72	.96	1.20	1.51	1.43	1.57	4.60
DX**Y	LINEAR(.1,10)**60	1.12	1.59	1.17	1.25	1.17	1.29	3.40
DX**Y	LOG(16**-65,16**63)**.7	1.11	1.57	1.16	1.27	1.16	1.29	3.39
CABS	POLAR(16**-16,16**16)	.92	1.85	1.32	1.84	1.65	1.87	5.40
CDABS	POLAR(16**-16,16**16)	.79	1.55	.97	1.34	1.20	1.42	3.64

10000 random arguments were generated according to the indicated distributions as defined above. The first column gives the percentage of arguments for which the function returns the correctly rounded result. The second column gives the average absolute error in ulps. Note that a perfect routine will have an average absolute error of 0.25 ulp if the residual errors are evenly distributed between −0.5 and 0.5 ulp. The third column gives the value of the 100th largest absolute error in ulps. (Note that we would expect the function to return a value in error by less than the amount in the third column 99% of the time.) For all columns we assume that the exact value is returned by the function of next higher precision. The tan and cotan functions in VFORTLIB and VALTLIB give an error return when close to a singularity (while ours compute a result). Cases where this occurs have been omitted in computing Columns 1–3 for these functions. It should be noted that although the VFORTLIB (and to a lesser extent the VALTLIB) functions appear to have very poor accuracy for the double-precision trigonometric functions on large arguments, a one-ulp error (even a rounding error) in a large argument produced by a user can cause a propagated error even in the exact result which is comparable to that generated by the corresponding VFORTLIB function.

Again, these tables should be taken as a general indication rather than a precise measurement of what performance (in terms of accuracy) the user can expect from the new library.

• *Ulp plots*

Figures 1–9 give ulp plots for three functions: DTAN in the range $-\pi/2$ to $\pi/2$ (Figs. 1–3), DSIN log distribution in the range $2^{-50}\pi$ to $2^{50}\pi$ (Figs. 4–6), and the short-precision X**Y, with $Y = 60.1$ and X linearly distributed in the range 0.1 to 10.0 (Figs. 7–9). The plots are given for all three libraries discussed in this paper: the new VS FORTRAN library (Figs. 1, 4, and 7), the old VS FORTRAN library (Figs. 2, 5, and 8), and VALTLIB or the Wang library (Figs. 3, 6, and 9). These are scatter plots with randomly distributed arguments along the x-axis and the corresponding ulp error along the y-axis. Note that when the

Table 6 Ratios of VALTLIB library measurements to Version 2 counterparts.

FUNCTION	DISTRIBUTION (RANGE)	4331-2	4361-5	4341-2	4381-3	3081KX	3090-S	3090-V
DCOS	LINEAR(-PI,PI)	2.09	2.55	2.01	2.30	1.96	1.48	2.88
DCOS	LOG(2**-50*PI,2**50*PI)	2.04	2.16	1.84	2.17	1.87	1.29	1.61
DSIN	LINEAR(-PI,PI)	2.15	2.65	2.03	2.43	2.07	1.44	3.02
DSIN	LOG(2**-50*PI,2**50*PI)	2.05	1.93	1.80	2.09	1.85	1.23	1.18
DTAN	LINEAR(-PI/2,PI/2)	1.71	2.00	1.81	2.04	1.79	1.38	1.99
DTAN	LOG(2**-39*PI,2**39*PI)	1.64	1.55	1.68	1.89	1.74	1.20	1.34
DCOTAN	LINEAR(-PI/2,PI/2)	1.72	2.02	1.83	2.08	1.80	1.42	2.03
DCOTAN	LOG(2**-39*PI,2**39*PI)	1.29	1.31	1.40	1.57	1.49	1.04	1.14
EXP	LINEAR(-100,100)	.81	2.09	1.39	2.62	1.93	1.84	5.18
EXP	LINEAR(-16,16)	.72	1.09	1.25	2.25	1.77	1.61	5.11
DEXP	LINEAR(-100,100)	1.16	1.15	1.18	1.24	1.28	1.13	3.84
DEXP	LINEAR(-16,16)	1.16	1.16	1.20	1.28	1.29	1.12	3.84
X**Y	LINEAR(.1,10)**60	.75	1.51	1.40	1.99	1.76	1.57	4.59
X**Y	LOG(16**-65,16**63)**.7	.75	1.35	1.41	2.01	1.76	1.57	4.60
DX**Y	LINEAR(.1,10)**60	1.27	1.64	1.32	1.58	1.49	1.67	4.41
DX**Y	LOG(16**-65,16**63)**.7	1.26	1.63	1.32	1.58	1.49	1.67	4.42

ulp error is large, it is plotted on the log scale (Figs. 2, 5, 6, and 8). The colored line represents 0.5 ulp error. For the linear ulp scale, all the points within the two colored lines represent correct rounding. For the logarithmic ulp scale, absolute value of the ulp error is plotted, and therefore all points below the colored line represent correct rounding.

The plots for the new library are essentially featureless, as most of the arguments are correctly rounded. The only noteworthy feature is in Fig. 4, where for small arguments the ulp error is essentially zero. For these arguments, $\sin(x) = x$ is a very good approximation, resulting in almost zero ulp errors. Although these plots look featureless on the present scale, they will show many features and peculiarities if magnified by a factor of 256 along the y-axis with appropriate magnification along the x-axis to show various table intervals.

The old VS FORTRAN library shows large errors due to argument reduction, as indicated in Fig. 2, near $\pm\pi/2$. As the argument approaches a multiple of $\pi/2$, the ulp error keeps increasing. The argument reduction error is amplified in Fig. 5, where we see a definite pattern. The staircase pattern is caused by the hexadecimal arithmetic, and the steps are a factor of 16 apart in both directions. The points above the staircase represent arguments near multiples of π. Another interesting feature of Fig. 5 is the spikes for small arguments, again at intervals of a factor of 16. These spikes are explained in the subsection on ulp plots in Section 2.

Next, we examine the equivalent plots for VALTLIB. In this library the argument reduction is done more accurately. But, as also in the old VS FORTRAN library, even for those arguments where the argument reduction is not needed, we find errors of up to 16 ulps, for DTAN in Fig. 3. The increased accuracy in argument reduction is more evident in Fig. 6, where the staircase phenomenon is delayed to much larger arguments. This indicates that more digits of π are used in the argument reduction. Even for this library, if DSIN or DTAN is examined for arguments very close to π, very large ulp errors would be observed. The rest of the features of Fig. 6, including the spikes for small arguments, are very similar to those of Fig. 5.

In the power function, when $y \log(x)$ is large, it is very important to calculate $\log(x)$ and $y \log(x)$ to a higher precision, to ensure good accuracy in the final result. This apparently was not the case with the old VS FORTRAN library, as indicated by Fig. 8. Figure 9 shows that this problem was considerably reduced in VALTLIB. Even for VALTLIB, experiments with very large exponents revealed errors of up to 100 ulps in short precision and several hundred ulps in long precision. The plot of Fig. 8 shows discontinuities very typical of hexadecimal arithmetic, when the result crosses to a new exponent. The new short-precision power function (Fig. 7) looks perfect, although it is not.

5. Conclusions

We have presented a description of a new practical theory for producing very accurate high-performance scalar and vector elementary functions for System/370 IBM machines. Our overall approach is to use tables in which we represent each function as an exact quantity plus a small correction. Although this idea is not new, we have shown, by a number of novel methods, that such an approach can be made extremely accurate and very fast. In contrast, the previous approach of using rational functions as a minimax approximation is *not* as accurate *nor* as fast. However, this older approach was developed when preservation of computer storage was important and the cost of a divide instruction compared to that of a multiply instruction was not large.

The development of the table approach followed rather naturally from the new requirement to produce vector algorithms. The requirement of producing bitwise-

Table 7 Cross-system comparisons. Column 1: VS FORTRAN Version 1 scalar times on 3081 versus Version 2 scalar on 3090. Column 2: VS FORTRAN Version 2 scalar times on 3090 versus Version 2 vector on 3090. Column 3: VS FORTRAN Version 1 scalar times on 3081 versus Version 2 vector on 3090.

FUNCTION DISTRIBUTION (RANGE)	3081K 3090S	3090S 3090V	3081K 3090V
EDUM	1.83		
E2DUM	1.75		
CDUM	1.79		
DDUM	1.76		
D2DUM	1.77		
CDDUM	1.72		
ACOS CIRC(0,PI)	4.72		
DACOS CIRC(0,PI)	5.09		
ASIN CIRC(-PI/2,PI/2)	4.54		
DASIN CIRC(-PI/2,PI/2)	5.23		
ATAN TAN(-PI/2,PI/2)	3.38	1.45	4.89
DATAN TAN(-PI/2,PI/2)	3.87	1.78	6.88
ATAN2 POLAR(16**-16,16**16)	3.05	1.67	5.10
DATAN2 POLAR(16**-16,16**16)	3.40	1.95	6.62
COS LINEAR(-PI,PI)	3.19	1.90	6.07
COS LOG(2**-18*PI,2**18*PI)	2.24	2.52	5.66
DCOS LINEAR(-PI,PI)	3.62	1.94	7.02
DCOS LOG(2**-50*PI,2**50*PI)	3.06	1.25	3.83
SIN LINEAR(-PI,PI)	2.86	2.01	5.74
SIN LOG(2**-18*PI,2**18*PI)	2.41	1.54	3.71
DSIN LINEAR(-PI,PI)	3.54	2.09	7.40
DSIN LOG(2**-50*PI,2**50*PI)	2.94	.96	2.82
TAN LINEAR(-PI/2,PI/2)	3.09		
TAN LOG(2**-18*PI,2**18*PI)	3.58		
DTAN LINEAR(-PI/2,PI/2)	3.10	1.44	4.46
DTAN LOG(2**-39*PI,2**39*PI)	2.68	1.12	2.99
COTAN LINEAR(-PI/2,PI/2)	3.18		
COTAN LOG(2**-18*PI,2**18*PI)	3.30		
DCOTAN LINEAR(-PI/2,PI/2)	3.20	1.43	4.59
DCOTAN LOG(2**-39*PI,2**39*PI)	2.33	1.09	2.54
EXP LINEAR(-100,100)	3.83	2.81	10.76
EXP LINEAR(-16,16)	3.32	3.18	10.56
DEXP LINEAR(-100,100)	2.86	3.42	9.75
DEXP LINEAR(-16,16)	2.82	3.44	9.72
ALOG LOG(16**-65,16**63)	3.12	2.72	8.47
DLOG LOG(16**-65,16**63)	3.18	3.24	10.31
ALOG10 LOG(16**-65,16**63)	2.67	3.19	8.52
DLOG10 LOG(16**-65,16**63)	2.83	3.12	8.82
SQRT LOG(16**-65,16**63)	2.50	3.16	7.91
DSQRT LOG(16**-65,16**63)	2.43	3.41	8.31
X**Y LINEAR(.1,10)**60	3.36	2.93	9.83
X**Y LOG(16**-65,16**63)**.7	3.35	2.93	9.80
DX**Y LINEAR(.1,10)**60	3.11	2.64	8.22
DX**Y LOG(16**-65,16**63)**.7	3.10	2.64	8.17
CABS POLAR(16**-16,16**16)	3.70	2.89	10.71
CDABS POLAR(16**-16,16**16)	2.78	2.56	7.13

compatible scalar algorithms that also performed well came about when we achieved the capability of viewing difficult and or special arguments as rare cases so that the main code stream had short paths with almost full vector lengths. These special situations are quickly detected by very inexpensive tests and handled in the scalar mode. At times we modified algorithms slightly to achieve better performance on the vector machine while maintaining the scalar compatibility; this was necessary because there is no vector counterpart for many scalar instructions. The relative timing of various instructions in vector and scalar mode is quite different, which leads to changes on a vector implementation. Some of the tables are somewhat expanded for the vector version to achieve better performance. We have exploited the fast vector multiply-add instruction wherever possible, especially in polynomial evaluations. In the vector mode, it is cheaper (on a per-element basis) to mask off underflow. This has been done for some functions to improve performance.

Acknowledgments

The authors have received help, advice, and support from many people. Throughout the project, S. Winograd has remained keenly interested and supportive and, in the early phase, contributed some detailed algorithms. At the outset, C. Micchelli, T. Rivlin, and R. Willoughby worked on the project; they offered us valuable early advice and assistance. Throughout the project we have profited from discussions with W. Cody, W. Kahan, A. Karp, G. Paul, and F. Ris. This is also true of C. Moler, who worked with IBM Palo Alto and to whom we are indebted for the ulp plot concept.

More recently, J. Cullum has worked closely with us in the transfer and description of our work. Initially D. Coppersmith read parts of the actual code and made some suggestions to improve it. D. Wells of Kingston helped design the vector interface and is responsible for the error-handling interfaces for the scalar and vector routines. We are also grateful to the many people in Kingston, Santa Teresa, Endicott, Haifa, and Böblingen who have worked on or have responsibility for the programming of elementary functions in IBM. From Kingston: J. Ruggiero, W. Heising, J. Ascoly, and A. Shannon; from Santa Teresa: J. Herman, A. Auyeung, R. Moyer, and J. Ehrman (who informed us of the early history of the FORTRAN intrinsic functions); from Endicott: D. Wehrly (who set the one-ulp standard and released the IBM products EML1 and EML2 [16]), F. Kozuh, T. Spillman, W. Perry, R. Headrick, and D. Jones; from Haifa: A. Hauber, J. Raviv, V. Amdurski, and S. Gal (the last two of whom told us about Gal's extra accurate-table method); from Böblingen: H. Bleher, D. Unkauf, and E. Lange. Some of the engineers of the 3090 were helpful in giving us timing information: R. Stanton, A. Vesper, L. Garcia, and S. Tucker. W. Buchholz was helpful in describing the vector architecture; A. Padegs and D. Gibson were supportive of our overall approach. E. Heeren and A. Jaffe, summer students at IBM Research, helped us analyze and exhaustively test our set of elementary functions.

References and notes

1. In this paper we use the term *subroutine* interchangeably with the correct term *FORTRAN intrinsic function.*
2. W. J. Cody, "Software for the Elementary Functions," *Mathematical Software*, J. Rice, Ed., Academic Press, Inc., New York, 1971, pp. 171–186.
3. W. J. Cody and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.
4. H. Kuki, "Mathematical Function Subprograms for Basic System Libraries—Objectives, Constraints, and Trade-Offs," *Mathematical Software*, J. Rice, Ed., Academic Press, Inc., New York, 1971, pp. 187–199.
5. W. J. Cody, "Software for the Elementary Functions," presented at the SIAM National Conference, Seattle, WA, July 1984.
6. H. Kuki and J. Ascoly, "FORTRAN Extended-Precision Library," *IBM Syst. J.* **10,** 39–61 (1971).
7. J. Y. Wang, "On the Improvement of Some Mathematical Subroutines in the IBM S/360 FORTRAN IV Libraries," SHARE-54, Anaheim, CA, March 3, 1980, Vol. 1, pp. 75–77.
8. J. Y. Wang and J. Boyer, "A Study of the Mathematical Routines in the IBM System/360 FORTRAN IV and FORTRAN IV (Mod II) Libraries," *AMD-TM 304*, Applied Mathematics Division, Argonne National Laboratory, January 1978.
9. J. Y. Wang, "The Evaluation of Periodic Functions with Large Input Arguments," *ACM/SIGNUM Newsletter* **13,** No. 4, 7–9 (1978).
10. Randolph G. Scarborough and Harwood G. Kolsky, "Improved Optimization of FORTRAN Object Programs," *IBM J. Res. Develop.* **24,** No. 6, 660–676 (1980).
11. C. Moler, "Mathematical Software for Vector Computers," presented at the SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, VA, November 10–11, 1983.
12. *Subroutine Library—Mathematics*, IBM Program Product 5736-XM7 (System/360 and System/370) 5711-XM2 (IBM 1130 and 1800); available through IBM branch offices.
13. A. Baker, *Transcendental Number Theory*, Cambridge University Press, Cambridge, England, 1975, p. 6.
14. G. Paul, IBM Research Division, Yorktown Heights, NY, private communication, October 1982.
15. F. Ris, IBM Research Division, Yorktown Heights, NY, private communication, October 1982.
16. *Elementary Math Library, Programming RPQ P81005 Program No. 5799-BTB Program Reference and Operations Manual:* a) First Edition (Release 1, EML1), January 1984, ST40-2230-00 (formerly SH20-2230-00); b) Second Edition (Release 2, EML2), August 1984, SH20-2230-1. Both are available through IBM branch offices.
    EML1 contained eight scalar programs, for short- and long-precision SQRT, EXP, LOG, LOG10; EML2 contains the 28 scalar programs described in this paper. All the programs described in this paper are contained in the VSF2LIB library of VS FORTRAN, Version 2.
17. T. E. Hull and A. Abrham, "Properly Rounded Variable Precision Square Root," *ACM Trans. Math. Software* **11,** No. 3, 229–237 (September 1985).
18. W. Kahan, "Software $\sqrt{x}$ for the Proposed IEEE Standard," Computer Science Department, University of California, Berkeley, August 1980; unpublished work.
19. W. Fullerton, private communication, February 1982.
20. S. Gal, "Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance," presented at the International Scientific Symposium of IBM Germany, March 12–14, 1985, Bad Neuenahr, Germany; proceedings to appear.
21. J. C. P. Miller, "Lecture Notes on Numerical Analysis," Cambridge University, Cambridge, England, 1958.
22. The vector routines produce unpredictable results for unnormalized nonzero arguments.

Received November 5, 1985; accepted for publication December 2, 1985

Ramesh C. Agarwal *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Agarwal received his B.Tech. degree (with honors) from the Indian Institute of Technology (IIT), Bombay, India, and the M.S. and Ph.D. degrees from Rice University, Houston, Texas, all in electrical engineering, in 1968, 1970, and 1974, respectively. During 1971–72, he was an Associate Lecturer at the School of Radar Studies, IIT Delhi, India; from 1974 to 1977 he was with the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. He spent the period 1977–1981 as a Principal Scientific Officer at the Centre for Applied Research in Electronics at IIT Delhi, India, and returned to IBM in 1982. His research interests have included network synthesis, information theory and coding, number theoretic transforms, fast algorithms for computing convolution and DFT, application of digital signal processing to structure refinement of large biological molecules using X-ray diffraction data, sonar signal processing, architecture for special-purpose signal processors, digital DTMF/MF receivers, filter structures, analysis of Kennedy assassination tapes, computation of elementary functions, and vectorization for engineering/scientific computations. Dr. Agarwal received the 1974 Acoustics, Speech, and Signal Processing Senior Award from the Institute of Electrical and Electronics Engineers for papers on number theoretic transforms, an IBM Outstanding Contribution Award in 1979 for work on crystallographic refinement of biological molecules, an IBM Outstanding Technical Achievement Award in 1984 for elementary functions work, and an IBM Outstanding Innovation Award in 1985 for his work in vectorizing the FFT algorithm.

James W. Cooley *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Cooley received his B.A. from Manhattan College, New York City, in 1949 and his M.A. and Ph.D. in applied mathematics from Columbia University, New York, in 1951 and 1961, respectively. He was a programmer on John von Neumann's electronic computer at the Institute for Advanced Study, Princeton, New Jersey, beginning in 1953; in 1956 he became a research assistant at the Computing Center of the Courant Institute at New York Unviersity, where he worked on numerical methods for quantum mechanical calculations. Since 1962, he has been on the Research staff of the Thomas J. Watson Research Center in Yorktown Heights, except for a one-year sabbatical, 1973–1974, which he spent at the Royal Institute of Technology, Stockholm, Sweden. At IBM he has worked on computational methods for solving diffusion and transport equations in applications to transistor and ionic flow problems. He has assisted in the development and use of mathematical models of the electrical activity in nerve and muscle membranes and in assorted eigenvalue problems and numerical methods for solving ordinary and partial differential equations. Dr. Cooley has been involved in the development of numerical methods for computers, including the fast Fourier transform and convolution algorithms. In recent years, he has participated in the development of software for elementary functions and signal processing programs for the IBM 3090 Vector Facility. He is a Fellow of the Institute of Electrical and Electronics Engineers. Dr. Cooley holds five IBM awards and four IEEE awards.

Fred G. Gustavson *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Gustavson received the B.S. degree in physics in 1957 and the M.S. and Ph.D. degrees in applied mathematics in 1960 and 1963, all from Rensselaer Polytechnic Institute, Troy, New York. From 1960 to 1961 and again from 1962 to 1963, he was engaged in research and teaching at Rensselaer Polytechnic Institute. From 1961 to 1962 he was with the U.S. Army Research Center at the University of Wisconsin. He joined the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center as a research staff member in 1963. During 1965 and 1966 he taught courses in the graduate division of Baruch University, New York. He is currently the manager of mathematical software in the Mathematical Sciences Department at Yorktown. His primary interest has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. He has worked in the area of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, elementary functions, and programming applications. Dr. Gustavson is a member of the Mathematical Association of America, Pi Mu Epsilon, Sigma Xi, and the Society for Industrial and Applied Mathematics. He received an IBM Outstanding Contribution Award for his work in sparse matrices and an IBM Outstanding Invention Award, jointly with R. K. Brayton and G. D. Hachtel, for the sparse tableau approach to network analysis and design. For the latter work he received the IEEE Circuit Theory Best Paper Award in 1971. In 1973 he received an IBM First Invention Achievement Award. In 1984 he received an IBM Outstanding Innovation Award for producing new highly accurate and significantly faster elementary-function algorithms for System/370 machines. In 1985, he received an IBM Outstanding Technical Achievement Award for his contribution to the design and implementation of novel new high-performance algorithms for solving linear equations.

James B. Shearer *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Shearer did his undergraduate work at the California Institute of Technology, Pasadena; he received his B.S. in 1976. In 1980 he received his Ph.D. from the Massachusetts Institute of Technology, Cambridge, for a thesis entitled "Some Problems in Combinatorics." Dr. Shearer joined IBM as a research staff member in the Mathematics Department in 1983. His research interests include combinatorics and mathematical software.

Gordon Slishman *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Slishman received the B.S. in mathematics in 1971 from Kansas State University, Manhattan. He joined IBM's Federal Systems Division in Owego, New York, in 1974, working as a VM and OS systems programmer; later he did real-time applications programming for LAMPS (Light Airborne Multi-Purpose System). In 1979, he joined the System Products Division in Endicott, New York, working as a designer of physical layout and logic for the printer channel adapter. In 1983 he worked as a programmer on the Elementary Math Library products EML1 and EML2 for the Systems Technology Division. He is at present an advisory programmer in the Mathematical Sciences Department at the Thomas J. Watson Research Center. Mr. Slishman has received an STD Award for software implementation of EML algorithms.

Bryant Tuckerman *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Tuckerman has been a mathematician at the IBM Thomas J. Watson Research Center since 1957. Prior to that he did military research (1941–45), taught at Cornell University, Ithaca, New York (1947–49), and at Oberlin College, Ohio (1949–52), and worked at the Electronic Computer Project of the Institute for Advanced Study (1952–57). He received his B.S. from Antioch College, Yellow Springs, Ohio, in 1939 and his M.A. (1946) and Ph.D. (1947) from Princeton University, New Jersey, all in mathematics. His work has included automatic programming; the computation of ancient planetary positions; character and speech recognition; arbitrary-precision integer arithmetic; computational number theory, including a lower bound for odd perfect numbers, and the discovery of the 24th Mersenne prime; speed-enhancing programming of computers; and cryptography, including analyzing the strengths of several cryptographic systems, devising the method of chaining which enhances the strength of DES (the Data Encryption Standard), and helping to implement IPS (Information Protection System), a software embodiment of DES with chaining. Besides working on these new scalar and vector elementary functions, he has also contributed to the matrix programs in the recently announced ESSL (Engineering and Scientific Subroutine Library), which has vector and scalar embodiments of a number of array-oriented functions. Dr. Tuckerman has received IBM Outstanding Innovation awards for his work on IPS and on these elementary functions, and a Research Division Award from Kingston for his work on ESSL.