Self-contained exponentiation*

by NANCY W. CLARK and W. J. CODY

Argonne National Laboratory
Argonne, Illinois

INTRODUCTION

The traditional implementation for floating-point exponentiation, \( x \) raised to the \( y \) power, is to compute \( \exp(y \ln(x)) \) using standard subroutines for the logarithm and the exponential function. While it is possible to provide extremely accurate subroutines for these latter functions, we shall shortly see that this is seldom done. Even in those rare cases where excellent subroutines are available, the exponentiation routine, for sound theoretical reasons, is poor. In this paper, we present brief statistics indicative of the quality of these three subroutines in the basic Fortran libraries provided by various manufacturers, a detailed error analysis for exponentiation, and a method for exponentiation via self-contained subroutines.

In the following discussion we will use the term exponentiation to refer to \( x^y \) where we will always assume \( x > 0 \). The term exponential will refer to \( e^x \) where \( e \) is a fixed constant base, usually either 2 or \( e \).

The present situation

With the cooperation of a number of different individuals and computing centers, we ran some simple tests on the exponential, logarithm and exponentiation subroutines in the basic Fortran libraries on eight different computers representing six different manufacturers. The only version of the single-precision library on the CDC-3600 available to us contained subroutines we had written according to the methods to be described and does not necessarily represent the manufacturer's library. We also tested our own version of the library for the IBM S/360 in addition to the standard library.

These tests were not intended to be complete certifications of the routines tested, but were designed to lightly probe areas where such subroutines are most likely to have trouble. The tests consisted of computations with a series of arguments exactly representable in binary notation. The corresponding function values were output in octal or hexadecimal form and compared against similar computations in 96-bit arithmetic on a CDC 6400. The computations involved were:

\[
\begin{align*}
\text{exp}(n) & \quad n = 40(1)88, \\
\ln(x) & \quad x = .25(.015625)2.0, \\
(2^n, 22 - n) & \quad n = 0(1)22, \\
(4^n, 11 - n/2) & \quad n = 1(1)22, \\
(2^n, 44 - 4n) & \quad n = 0(1)11, \\
(.75 \times 2^n, 46 - 4n) & \quad n = 1(1)11.
\end{align*}
\]

The test results are summarized in Table I.

Certain of the computers used have either octal or hexadecimal floating-point arithmetic. On these computers, a mantissa can be properly normalized and still have the first two or three bits zero. This accounts for the apparent tabular discrepancies between the sum of the maximum number of bits in error and the minimum number of correct bits, and the total number of bits in the mantissa on these machines.
**TABLE I—Accuracy Test Results**

<table>
<thead>
<tr>
<th>Machine and Subroutine</th>
<th>Single-Precision</th>
<th>Double-Precision</th>
<th>Machine and Subroutine</th>
<th>Single-Precision</th>
<th>Double-Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td>N</td>
<td>M</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>Burroughs B-5000</td>
<td>(39 bit mantissa)</td>
<td>(78 bit mantissa)</td>
<td>IBM 360/75 IBM library</td>
<td>(24 bit mantissa)</td>
<td>(56 bit mantissa)</td>
</tr>
<tr>
<td>EXP</td>
<td>9</td>
<td>30</td>
<td>8</td>
<td>69</td>
<td>EXP</td>
</tr>
<tr>
<td>LN</td>
<td>3</td>
<td>35</td>
<td>7</td>
<td>71</td>
<td>LN</td>
</tr>
<tr>
<td>X**Y</td>
<td>7</td>
<td>31</td>
<td>11</td>
<td>67</td>
<td>X**Y</td>
</tr>
<tr>
<td>Control Data 3600</td>
<td>(36 bit mantissa, Argonne library)</td>
<td>(84 bit mantissa, CDC library)</td>
<td>IBM 360/75 Argonne library</td>
<td>(24 bit mantissa)</td>
<td>(56 bit mantissa)</td>
</tr>
<tr>
<td>EXP</td>
<td>1</td>
<td>35</td>
<td>4</td>
<td>80</td>
<td>EXP</td>
</tr>
<tr>
<td>LN</td>
<td>2</td>
<td>34</td>
<td>5</td>
<td>79</td>
<td>LN</td>
</tr>
<tr>
<td>X**Y</td>
<td>1</td>
<td>35</td>
<td>8</td>
<td>76</td>
<td>X**Y</td>
</tr>
<tr>
<td>Control Data 6400</td>
<td>(48 bit mantissa)</td>
<td>SDS Sigma 7 (24 bit mantissa)</td>
<td>(56 bit mantissa)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td>1</td>
<td>47</td>
<td>EXP</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>LN</td>
<td>2</td>
<td>46</td>
<td>LN</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>X**Y</td>
<td>7</td>
<td>41</td>
<td>X**Y</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>G.E. 225</td>
<td>(30 bit mantissa, FIZMOP system)</td>
<td>Univac 1107 (27 bit mantissa)</td>
<td>(54 bit mantissa)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td>3</td>
<td>27</td>
<td>EXP</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>LN</td>
<td>12</td>
<td>18</td>
<td>LN</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>X**Y</td>
<td>10</td>
<td>20</td>
<td>X**Y</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>G.E. 645</td>
<td>(27 bit mantissa)</td>
<td>(63 bit mantissa)</td>
<td>Univac 1108 (27 bit mantissa)</td>
<td>(60 bit mantissa)</td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td>1</td>
<td>26</td>
<td>14</td>
<td>49</td>
<td>EXP</td>
</tr>
<tr>
<td>LN</td>
<td>4</td>
<td>23</td>
<td>4</td>
<td>59</td>
<td>LN</td>
</tr>
<tr>
<td>X**Y</td>
<td>1</td>
<td>26</td>
<td>14</td>
<td>49</td>
<td>X**Y</td>
</tr>
</tbody>
</table>

M = maximum number of bits in error.  
N = minimum number of correct significant bits.

We will show presently that accuracy in exponentiation depends very heavily on the accuracy in the calculation of the exponential function. Note, however, that even with a good exponential function, as is apparently the case in the single precision CDC 6400 and the original IBM 360 libraries, the exponentiation routine can still be in error by two to three significant decimal places or more. Also note that the exponentiation routines corresponding to our methods as well as the single-precision routine on the G.E 645 display primarily round-off error in these tests.

**Error analysis**

There are two major types of error in any function subroutine. The first is transmitted error, i.e., error due to small errors in the arguments. If we assume

\[ z = f(x) \]

where \( f(x) \) is differentiable, then

\[ \delta z \approx x \frac{f'(x)}{f(x)} \delta x \] (1)

where

\[ \delta z = \Delta z / z \approx dz / z \] (2)

denotes the relative error in \( z \), and \( \Delta z \) denotes the absolute error in \( z \). It is clear that the transmitted error, \( \delta z \), depends solely on the inherited error, \( \delta z \), and not on the subroutine. The second type of error is generated error, i.e., that error generated by the...
computational process. This includes both errors due to truncating an infinite process at some finite point and roundoff errors.

Even infinitely precise subroutines have no control over inherited error. Therefore, in designing subroutines we assume there is no inherited error and seek to minimize the generated error.

Now let us consider the logarithm-exponential method for exponentiation. We use the relation

\[ x^w = e^w, \quad x > 0, \tag{3} \]

where

\[ w = y \log_c(x). \]

From (1) and (2), and recalling our assumption that \( \delta x = \delta y = 0 \), we see

\[ \Delta w = y \Delta s \]

where \( \Delta s \) represents only the generated error from the logarithm computation.

If

\[ u = e^w, \]

then

\[ \delta u = \ln e \Delta w + \delta G(w) \tag{4} \]

where \( \delta G(w) \) denotes the generated relative error from the exponential computation. For good exponential routines \( \delta G(w) \) affects only the least significant one or two bits of \( u \). Thus, the relative error in the exponentiation is essentially proportional to the absolute error in \( w \). Clearly, we want to minimize \( \Delta w \) as it appears to the exponential routine.

There are two major contributions to this error: the generated error from the logarithm calculation, and the finite word length of the computer. The second is by far the more important of the two. Suppose the floating-point mantissa of the calculator contains \( 2t \) significant bits, but \( w \) is of the order of \( 2^t \). Then the floating-point representation of \( w \), the argument to be passed to a standard exponential routine, may have a rounding error as large as \( 2^{-t} \), i.e., \( \Delta w \approx 2^{-t} \). Consequently, \( u \) may be accurate to only about \( t \) bits independently of the accuracy of the logarithm calculation. This is the reason some of our tests found inaccurate exponentiation even though the logarithm and exponential routines appeared to be reasonably accurate.

**A new approach**

There are at least two alternatives to the traditional computation. One is to resort to “overkill” by carrying out the traditional computation in a higher precision arithmetic. This is expensive in time; it is easy to do for single-precision routines, but difficult for double precision routines. (Is this the approach on the G. E. 645?) The second alternative is to raise the status of exponentiation routines. At the moment they are considered to be secondary routines which call upon the primary routines for the exponential and logarithm. We propose that they become primary, self-contained routines with possible secondary entry points for the exponential and logarithm.

If we accept this major reversal in philosophy, we free the computation of several restrictions. For example, we need not pick \( c = e \) in Eqs. (3) and (4), but can make the choice \( c = 2 \) which appears most natural for a computer, and which introduces the factor \( \ln 2 = 0.69315 \) in Eq. (4). This permits us to obtain extra significance in the results of the logarithm computation, as we shall shortly see, and to retain this significance throughout the remainder of the calculation.

The first implementations of the algorithm we will outline were programmed using single-precision fixed-point arithmetic to do single-precision exponentiation on both the CDC 3600 and the IBM 360 computers. Because neither computer allows efficient double-precision fixed-point arithmetic, the algorithm has to be modified to use double-precision floating-point arithmetic to do double-precision exponentiation. So that the presentation will not be too abstract, we will present basically the algorithm as used on the IBM 360 in double-precision. Modifications for single-precision floating point or fixed point versions, or for other machines should be obvious.

We first reduce the range over which the logarithm must be approximated. Let

\[ x = 2^m \cdot m, \quad 1/2 \leq m < 1, \]

and choose

\[ b = n/16 \]
and

\[ a = 2^{-s/16}, \]

where \( n \) is an odd positive integer less than 16, such that

\[ x = 2^{k-b} m/a \]

where

\[ |\log_2(m/a)| \leq 1/16. \]

Then

\[ s = \log_2(x) = s_1 + s_2 \]

where

\[ s_1 = k - b, \]

\[ s_2 = \log_2 \left( \frac{1 + z}{1 - z} \right), \]

\[ \frac{1 + z}{1 - z} = \frac{m}{a}, \]

and

\[ z = \frac{m - a}{m + a}. \]

Since \( z \) is quite small (\(|z| \leq 0.022\)), \( s_2 \) is easily computed to full floating-point accuracy using a low order rational approximation, or even the first few terms of the Taylor series, provided \( z \) is computed accurately. (A little extra care is necessary at this point in base 16 floating-point, but we will not go into the details here.) Since \( x \) is assumed to be exact, \( m \) is exact and we can achieve full precision in \( m-a \) by breaking the constant \( a \) into two parts such that

\[ a = a_1 + a_2 \]

to the precision desired and such that the exponent on \( a_2 \) is much less than that on \( a_1 \). Then the computation

\[ m - a = (m - a_1) - a_2 \]

will retain the low order bits of \( a \). Normal floating-point can be used for the rest of the evaluation of \( z \).

Note that by carrying \( s_1 \) as one floating point number, and \( s_2 \) as another, we have rather painlessly achieved a logarithm accurate to well beyond usual working precision. Since \(|s_2| \leq 1/16\), the absolute error in \( s \) is now about \( 2^{-1} \) times the normal relative error in floating point. Careful multiplication of \( s \) by \( y \) will minimize the crucial quantity \( \Delta w \). At this point, the usefulness of fixed-point arithmetic with the extra significant bits in the representation of a number is apparent. When such arithmetic is not available, as we have assumed is the case, it is necessary to arrange the floating-point computations to achieve the extra significance at minimal cost. This is done as follows.

Let us say we reduce a number \( z \) when we write it in the form

\[ z = z_1 + z_2 \]

such that \( z_1 \) is the integer part of \( 16z \). Essentially, then, \( z \) is already in reduced form. We compute the exponent \( w \) in reduced form by writing

\[ y = y_1 + y_2, \]

where \( y_1 \) and \( y_2 \) are the double-precision representations of the most significant and least significant halves of \( y \) respectively, and forming the products \( s_1y_1, s_2y_1, \) and \( s_1y_2 \). Each of these quantities is again reduced and the results combined to form the reduced

\[ w = w_1 + w_2. \]

Now \( w_1 \) is of the form

\[ w_1 = \ell + j/16 \]

where \( \ell \) and \( j \) are integers. We then finally compute the exponential value

\[ u = 2^{\ell} \times 2^{j/16} \times 2^{a_1}. \] (5)

Since \(|w_2| \leq 1/16\), a Taylor series computation of the exponential is quite efficient, although we used rational Chebyshev approximations. The quantities \( 2^{j/16} \) can be carried in a table. In fact, if Eq. (5) is rewritten as

\[ u = 2^{\ell+1} \times 2^{(j-16)/16} \times 2^{a_2} \]

and the quantities \( 2^{-n/16} \) are tabulated, the same table can be used for the constant \( a \) needed in the logarithm computation. This dictates the form of the earlier decomposition of \( a \) into \( a_1 \) and \( a_2 \). Clearly \( a_1 \) should be the value of \( a \) correctly rounded to working precision while \( a_2 \) becomes a positive or negative correction term.
TABLE II—Random argument tests on conventional double-precision $X^{*}Y$ on IBM 360/75

<table>
<thead>
<tr>
<th>Argument Range</th>
<th>Frequency of Bit Errors</th>
<th>Max. Rel. Error</th>
<th>RMS Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of bits in error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$ $y$</td>
<td>0 1 2 3 4 5 6 7 8 9 10</td>
<td>100 other</td>
<td></td>
</tr>
<tr>
<td>$(1/16,16)$ $(-4,4)$</td>
<td>272 467 405 371 240 197 47 1 0 0 0 0</td>
<td>1.25E-15 3.65E-16</td>
<td></td>
</tr>
<tr>
<td>$(2^{-16},2^{16})$ $(-16,16)$</td>
<td>78 123 153 168 247 377 321 294 105 44 0 0</td>
<td>8.82E-15 2.70E-15</td>
<td></td>
</tr>
<tr>
<td>$(2^{-32},2^{32})$ $(-8,8)$</td>
<td>80 109 131 152 216 288 295 234 241 120 86 48</td>
<td>5.08E-14 9.60E-15</td>
<td></td>
</tr>
<tr>
<td>$(2^{-64},2^{64})$ $(-4,4)$</td>
<td>57 95 115 126 161 215 293 352 303 192 82 9</td>
<td>2.68E-14 6.97E-15</td>
<td></td>
</tr>
<tr>
<td>$(2^{-8},2^{8})$ $(-32,32)$</td>
<td>59 90 115 109 199 312 406 343 253 107 7 0</td>
<td>1.40E-14 4.02E-15</td>
<td></td>
</tr>
<tr>
<td>$(1/16,16)$ $(-64,64)$</td>
<td>60 96 110 128 196 275 318 318 281 167 48 3</td>
<td>1.95E-14 5.73E-15</td>
<td></td>
</tr>
</tbody>
</table>

Average execution time for $(x, y)$ random in $(0, 1) = 195 \mu$secs.

TABLE III—Random argument tests on self-contained double-precision $X^{*}Y$ on IBM 360/75

<table>
<thead>
<tr>
<th>Argument Range</th>
<th>Frequency of Bit Errors</th>
<th>Max. Rel. Error</th>
<th>RMS Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of bits in error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$ $y$</td>
<td>0 1 2 3 4 5 6 7 8 9 10</td>
<td>100 other</td>
<td></td>
</tr>
<tr>
<td>$(1/16,16)$ $(-4,4)$</td>
<td>1301 677 22 0 0 0</td>
<td>2.22E-16 6.24E-17</td>
<td></td>
</tr>
<tr>
<td>$(2^{-16},2^{16})$ $(-16,16)$</td>
<td>1206 759 35 0 0 0</td>
<td>2.22E-16 6.11E-17</td>
<td></td>
</tr>
<tr>
<td>$(2^{-32},2^{32})$ $(-8,8)$</td>
<td>1314 667 19 0 0 0</td>
<td>2.22E-16 5.81E-17</td>
<td></td>
</tr>
<tr>
<td>$(2^{-64},2^{64})$ $(-4,4)$</td>
<td>1350 634 16 0 0 0</td>
<td>2.21E-16 5.44E-17</td>
<td></td>
</tr>
<tr>
<td>$(2^{-8},2^{8})$ $(-32,32)$</td>
<td>1097 812 89 2 0 2</td>
<td>2.22E-16 6.31E-17</td>
<td></td>
</tr>
<tr>
<td>$(1/16,16)$ $(-64,64)$</td>
<td>872 823 250 52 3</td>
<td>2.22E-16 6.94E-17</td>
<td></td>
</tr>
</tbody>
</table>

Average execution time for $(x, y)$ random in $(0, 1) = 180 \mu$secs.

Since the last two factors in $u$ are each less than unity in magnitude, and the $2^{t+1}$ factor affects only the floating-point exponent, we see that the construction of $u$ from its factors is a stable process. Note that the error $\Delta w$, hence by Eq. (4) the error $\Delta u$, now depends, primarily on the magnitude of $y$. Using Eq. (4), and noting that we have gained an extra four bits in our calculation of $s$, we see that $y$ must be greater than roughly 32 before the inaccuracies in $w$ become large enough to greatly affect $\Delta u$. To verify this point, and to provide an in-depth comparison of our method and of the traditional computation, we have subjected our routine for the IBM 360 and the original IBM routine to a full certification as described in references one and two. The results, for identical tests, are presented in Tables II and III.

One final word about the fixed point version of this algorithm. In fixed point, the extra bits over $t'$ normal floating point mantissa length are already available. As we have indicated, the decomposition of $a$ and $y$ and the reduction of $s$, $w$, etc. are no longer necessary. This constitutes a savings in storage as well as in the number of instructions to be executed. But no matter which approach is taken, the fixed point or the floating point, the self-contained routine can be expected to be competitive timewise with the traditional routine because we have saved the overhead of linking with other subroutines. All three of the self-contained programs we have written are actually faster than their traditional counterparts. The price is paid in terms of storage. This price can be minimized by incorporating entries for the exponential and logarithm routines into the exponentiation routine, thus eliminating separate routines for the former.
ACKNOWLEDGMENTS

We would like to acknowledge the assistance of J. Boyle, P. Businger, L. Fosdick, C. Hammer, J. Pagels, R. Royston, H. C. Thacher, Jr., J. F. Traub and the computing centers at Argonne, Bell Telephone Laboratories, University of Illinois, Northwestern University, Univac, and the University of Notre Dame for their assistance in the tests reported in Table I.

REFERENCES

1  W J CODY
   Performance testing of function subroutines
   Proc SJCC 1969 759-763

2  N CLARK  W J CODY  K E HILLSTROM
    E A THIELEKER
   Performance statistics of the Fortran IV(H) library for the
   IBM system/360
   Report ANL 7321 Argonne Natl Lab 1967