The influence of machine design on numerical algorithms*

by W. J. CODY
Argonne National Laboratory
Argonne, Illinois

INTRODUCTION
A great deal has been said and written recently about the poor arithmetic characteristics of computer designs, largely because anomalies long present in the floating point arithmetic of binary computers have suddenly been magnified in importance by the appearance of base 16 arithmetic. This paper is not intended as an attempt to influence present or future machine design—we will leave that project to those who think they can influence such designs. Rather, it is written in the spirit of H. R. J. Grosch when he advised programmers "to go back to work; quit trying to remodel the hardware ... accept reality..."6

It is our purpose to examine the "reality" of some of the most common design features of recent computers and how they affect the implementation of numerical algorithms. To do this we will consider the floating point arithmetic units on two commercial computers, the Control Data 3600 and the IBM System/360, used by the Applied Mathematics Division at Argonne National Laboratory. These two machines are conventional in design, yet offer great variety and contrast in those arithmetic design characteristics we wish to discuss.

Both machines have built-in single and double precision floating point. The CDC machine, however, operates with exponent base 2 while the IBM machine uses exponent base 16. Of course, both machines treat the mantissa separately from the exponent in floating point operations, but the 3600 rounds the mantissa prior to a renormalization shift while the 360 truncates prior to renormalization in single precision. Renormalization shifts in the 3600 are in multiples of one bit while those in the 360 are in multiples of four bits. Thus, after multiplication of two normalized numbers the 3600 has at most a one bit shift for renormalization while the 360 shifts either zero or four bits.

There are other aspects of these machines that are important in detailed numerical work but we will concern ourselves only with the availability of double precision, the effects of rounding (including number truncation) and the renormalization shift.

Effect of renormalization
One of the first things one must realize in detailed floating point numerical computation is that computers deal with a finite discrete subset of the real number system using arithmetic that does not always obey some of the usual rules; e.g., the associative and distributive laws may not hold. Most of the trouble occurs in the process of truncation or rounding of results coupled with possible renormalization shifts.

As an indication of what can happen, let us show that three of the four possible modes of floating point arithmetic on the computers under discussion do not allow a multiplicative identity in the mathematical sense of the term. For illustrative purposes, assume the arithmetic is single precision as incorporated in the CDC 3600, but assume only four bits in the mantissa of a number. Let \( x = 1.125 \) and consider the product \( 1.0 \times x \). Then (the mantissas below will be expressed as binary fractions)

\[
\begin{align*}
x &= 2^4 \times \text{.}1001 \\
x \times 1.0 &= 2^5 \times \text{.}0100 \\
&= 2^2 \times \text{.}01001
\end{align*}
\]

The result mantissa is now rounded to give

\[
2^2 \times \text{.}0101
\]

and then renormalized by a left shift of one unit with an appropriate adjustment of the exponent to give

\[
2^1 \times \text{.}1010 = 1.250.
\]
Thus this commonly used rounding scheme when employed prior to the renormalization shift precludes the existence of a multiplicative identity.

If the rounding process were replaced by simple truncation, but still employed prior to renormalization (as happens on the CDC machine if rounding is suppressed and on the IBM machine in double precision), the above result would be

\[ 2^4 \times .1000 = 1.000. \]

Again unity is not a true multiplicative identity. Only the IBM single precision mode with its four guard bits and truncation after renormalization guarantees that

\[ 1.0 \times x = x \]

for all \( x \).

The important point here is not the lack of a multiplicative identity in the mathematical sense, but that predictable arithmetic errors have occurred. A little thought will show that the same erroneous results can occur any time one of the factors is an exact power of 2. However, in the CDC machine the error is always limited to the last bit while it will involve the last four bits when it occurs on the System/360. The replacement of the expression 2.0 \( \times \) \( x \), for instance, by \( x + x \) saves on the average about \( \frac{1}{2} \) bit per operation on the CDC machine, but saves almost 2 bits per operation on the IBM machine in double precision. Thus the discrepancy present in most binary machines is magnified by the use of base 16 arithmetic.

The use of four guard bits in the IBM single precision mode effectively corrects the problem of renormalization in multiplication. Note, however, that single and double precision arithmetic no longer obey the same rules, hence there may be Fortran programs, for example, in which conversion from one precision to another will alter the arithmetic behavior.

Consider next the use of Newton-Raphson iteration for the square root. Let \( A \) be the square root of \( x \), let \( A_n \) be an initial estimate of \( A \) and use the usual iteration scheme

\[ A_n = \frac{1}{2}(A_{n-1} + \frac{x}{A_{n-1}}) \quad n = 1, 2, \ldots \]

to obtain successively better estimates. This is known to be a convergent scheme with \( \{A_n\} \) approaching \( A \) monotonically from above for \( n > 0 \). If the iteration is continued until two successive estimates are equal, no further improvement is theoretically possible and we have the best estimate of \( A \). But let us follow the iteration in detail with System/360-type arithmetic (we will use only 12 bits in the mantissa) assuming we are near convergence and \( A_{n-1} \) has a mantissa greater than \( \frac{1}{2} \). Say

\[ A_{n-1} = 16^m \times .1010 1011 0000 + x/A_{n-1} = 16^m \times .1010 1010 1010 \]

\[ 16^m \times 1.0101 0101 0101. \]

The mantissa is now shifted right four places to give

\[ 16^{m+4} \times .0001 0101 0101 \]

and the last four bits are truncated. (There go the single precision guard bits!) Now multiplication by \( \frac{1}{2} \) gives

\[ 16^{m+5} \times .0011 0010 0010 \]

\[ \times x = 16^m \times .0000 0000 0000 \]

\[ 16^{m+3} \times .0000 1010 1010 1010 \]

which becomes

\[ 16^m \times .1000 0101 0101 \]

upon renormalization if single precision is used, and

\[ 16^m \times .1010 1010 0000 \]

if double precision is in use. Note that the last three bits are of necessity zero in single precision in spite of the existence of the guard bits. Not only are more accurate estimates of \( A \) possible, but it may be that \( A > A_n \) with this method, contrary to the theory.

Rewriting the iteration in the form

\[ A_n = A_{n-1} + \frac{1}{2}(x/A_{n-1} - A_{n-1}) \]

for at least the final iteration clears up the problem.

The original iteration scheme works quite well on the CDC machine, since the normalization shift after the addition involves only one bit and the multiplication by \( \frac{1}{2} \) is accomplished with no error by an alteration of the exponent in the results.

**Truncation vs Rounding**

As we have already seen, the four guard bits in the System/360 single precision operations offer protection in the case of multiplication. They offer no protection, however, against the build-up of truncation errors, even in short computations. Although this phenomenon is well known the following example of its effect may be of some interest to those who have never before witnessed it.

Consider the computation

\[ y = \int_0^1 x/3 \, dx \]

by means of trapezoidal quadrature using a mesh size \( h = 1/n, n = 2, 4, 8, \ldots, 256 \). That is, approximate \( y \) by the computation

\[ h \sum_{j=0}^n \frac{x}{3} \]

From the collection of the Computer History Museum (www.computerhistory.org)
where the double prime indicates only half of the first and last terms are to be considered. It will become obvious that one would never carry out this particular computation in the manner we will describe, nor for the large values of n we will use. The justifications for this example are that it is not far from being a practical problem (one need only change the integrand) and it is possible to almost completely isolate the effect of truncation.

We will actually construct the \((j \times h)/3\) for the various values of j and sum them. For the choices of h made here the quantities \(j \times h\) can always be expressed exactly as machine numbers, eliminating one possible source of error. Theoretically trapezoidal quadrature should give an exact answer for a linear integrand, eliminating a second source of error. Finally, if the integrand is computed in double precision as needed and then converted to single precision, the only remaining sources of error are possible inaccuracies in conversion to single precision and the summation process itself. Table I lists the results of the computation carried out with Fortran codes in single precision arithmetic, with the exception noted above, on both machines. The results are listed in octal and hexadecimal format to avoid the error contamination involved in conversion to decimal form. Note that the results on the CDC machine never stray too far from the correct results even when \(n\) becomes fairly large, whereas the System/360 results become progressively worse as \(n\) increases. Repeating the experiment in complete double precision, nullifying the possible effect of computation of the integrand in a higher precision, made no essential change in the low order bit patterns.

**TABLE I**

Trapezoidal Quadrature Applied to \(\int_0^{x/3} dx\) in Single Precision Floating Point on CDC 3600 and IBM System/360

<table>
<thead>
<tr>
<th>No. of Points</th>
<th>CDC 3600</th>
<th>IBM System/360</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1775525252525254</td>
<td>402AAAAA</td>
</tr>
<tr>
<td>4</td>
<td>1775525252525254</td>
<td>402AAAAA</td>
</tr>
<tr>
<td>8</td>
<td>1775525252525252</td>
<td>402AAAA8</td>
</tr>
<tr>
<td>16</td>
<td>1775525252525254</td>
<td>402AAAA8</td>
</tr>
<tr>
<td>32</td>
<td>1775525252525252</td>
<td>402AAAA7</td>
</tr>
<tr>
<td>64</td>
<td>1775525252525254</td>
<td>402AAAA6</td>
</tr>
<tr>
<td>128</td>
<td>1775525252525252</td>
<td>402AAAA9A</td>
</tr>
<tr>
<td>256</td>
<td>1775525252525252</td>
<td>402AAAA87</td>
</tr>
<tr>
<td>Correct Value</td>
<td>1775525252525253</td>
<td>402AAAAA</td>
</tr>
</tbody>
</table>

For this particular example the IBM arithmetic shows up quite badly. In view of the short word length (24 bit mantissa) in single precision it is particularly important to use an algorithm that will minimize the accumulation of truncation error. Examples can be concocted which show that rounding errors can also build up quite badly, but rounding is a much less biased operation than truncation and is therefore not as likely to cause serious deterioration of results. It is also the author's feeling that effective remedial action on the part of the programmer, e.g., the rearrangement of computations so as to minimize error build-up, is easier when the machine arithmetic uses rounding than it is when the arithmetic uses truncation.

**Double precision**

One feature of recent computers has simplified rather than complicated the work of a numerical analyst—the inclusion of inexpensive (timewise) double precision arithmetic. The advantage of double precision accumulation of inner products in certain matrix problems is well known, as is the use of double precision accumulation of the corrections to the dependent variable in certain methods for the solution of differential equations. But there is one rather important usage of double precision which has been neglected in the literature.

Let us consider the design of a subroutine for the exponential function

\[ y = e^x. \]

We recognize at the start that an argument fed to this routine will probably be in error because it is a machine number of finite length. Let \(\Delta y\) be the absolute error in \(y\), and

\[ \delta y = \Delta y/y \]

be the relative error. For the exponential function,

\[ \delta y = x \delta x \]

and a rounding error of \(\frac{1}{2}\) unit in the least significant bit of \(x\) for \(x = 80\) can be expected to lead to an error of 40 units in the least significant bit of \(y\). Many exponential routines yield errors of this order of magnitude even when the value of \(x\) is exactly 80, i.e., even when \(\delta x = 0\).

The inaccuracy lies in the argument reduction scheme usually used. We will assume we are using the CDC machine with its base 2 arithmetic in what follows, although this assumption is not crucial. A typical exponential routine then proceeds somewhat as follows. Let

\[ n = \lfloor x/n \rfloor 2 + \frac{1}{2} \]
where \([a]\) denotes the “integer part of \(a\),” and let
\[g = x - n \cdot \lfloor n/2 \rfloor.\]

Then
\[e^x = 2^n \cdot e^g,\]

reducing the problem of computing any exponential to that of computing the exponential of a reduced argument \(g\) where \(|g| \leq \lfloor n/2 \rfloor/2\). This last can generally be done quite accurately provided \(g\) is accurate. The greatest loss in accuracy occurs in the subtraction involving the two nearly equal quantities \(x\) and \(n \cdot \lfloor n/2 \rfloor\). When these quantities agree for the first \(m\) significant bits, \(g\) may be in error in the last \(m\) significant bits leading to a large value of \(6g\) independently of whether \(6x\) is zero or not.

It is precisely here that double precision operations can save the day. Assume \(6x = 0\), hence \(x\) can be converted to a double precision number with no introduced error. Since \(n \cdot \lfloor n/2 \rfloor\) is a constant known to any precision desired, the product \(n \cdot \lfloor n/2 \rfloor\) can be found in double precision and the value of \(g\) determined quite accurately at a total cost of one double precision multiplication and one double precision subtraction. The cost of converting the single precision variables \(x\) and \(n\) to double precision and of converting \(g\) back again is generally quite small when programming is done in assembly language. This last conversion could even be simple truncation of the mantissa. Since \(|g| < .7\), an error of 1 unit in the last bit of \(g\) will propagate at most as .7 unit in the last bit of \(e^g\).

The payoff for this technique can be quite large. Table II compares the results of computations of \(e^x\) for \(x = 80(1)100\) with routines differing only in the argument reduction scheme. In a systems library the accuracy obtainable in this manner is of utmost importance. Not only do users regard systems routines as “black boxes” which return correct results, but other systems routines do as well. It is not at all uncommon for an exponential routine to be called by the system routines for complex trigonometric functions as well as by the power routine, i.e., the routine to evaluate the Fortran expression \(x**y\). The basic CDC 3600 library developed at Argonne National Laboratory uses this argument reduction technique in almost all of the real and complex trigonometric routines as well as the power routine. Computations with these routines rival in accuracy computations made by converting the independent variable to double precision and using the appropriate double precision routine. It is generally even more accurate to evaluate \(x**1\) by converting \(I\) to floating point and using the floating point power routine than it is to use the integer power routine directly.

Single precision floating point operations that produce double length results, as in the CDC machine, can also be quite useful. Consider the computation of

\[
\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt.
\]

Table II

| Error in units of least significant bit for computation of \(e^x\) on a CDC 3600 in single precision floating point. An Error of -20 indicates the computed result was 20\(_0\) units low |
|---------------------------------|-------------------------------------------------------------|
| Error using standard argument reduction | Error using modified argument reduction |
| \(x\) | \(e^x\) | \(x\) | \(e^x\) |
| 80 | 48 | 0 | 0 |
| 81 | 54 | 0 | 0 |
| 82 | 6 | -1 | 0 |
| 83 | -16 | 0 | 0 |
| 84 | -28 | 0 | 0 |
| 85 | 76 | 1 | 1 |
| 86 | 35 | 0 | 0 |
| 87 | 27 | -1 | 0 |
| 88 | 7 | -1 | 0 |
| 89 | -16 | -1 | 0 |
| 90 | -47 | 0 | 0 |
| 91 | 57 | 0 | 0 |
| 92 | 54 | 1 | 1 |
| 93 | 19 | 0 | 0 |
| 94 | 3 | 1 | 1 |
| 95 | -13 | 0 | 0 |
| 96 | 85 | -1 | 0 |
| 97 | 87 | 0 | 0 |
| 98 | 39 | -1 | 0 |
| 99 | 27 | 0 | 0 |
| 100 | 0 | 0 | 0 |

for large values of \(x\) by means of the asymptotic expansion

\[
\text{erfc}(x) \approx \frac{1}{\sqrt{\pi}} \frac{e^{-x^2}}{x} \left[ 1 + \sum_{m=1}^{\infty} \frac{(2m-1)!!}{(2x^2)^m} \right]
\]

where \((2m-1)!! = \frac{(2m)!}{2^m(m)!}\) for \(m > 0\), \(1.3.5\ldots(2m-1) = 1\) for \(m = 0\).

For \(x\) large enough the main source of error lies in the evaluation of \(e^{-x^2}\). Even if \(x\) is considered to be exact, i.e., \(6x = 0\), the product \(x^2\) used as an argument for the single precision exponential routine has an error in the least significant bit due to rounding. Double
precision argument reduction within the exponential routine cannot save the computation, but an analogous reduction before entry to the exponential routine can. The quantity \( x^2 \) can be formed accurately to double precision by either converting \( x \) to double precision and squaring, or by squaring \( x \) in single precision while suppressing the rounding and re-normalization of the double length product. Computation of the reduced argument \( g \) can then proceed as outlined above, and it is this reduced argument which is transmitted to the exponential routine.

SUMMARY

Through the use of some elementary examples we have shown that the design of the arithmetic unit of a computer can influence the design of accurate numerical subroutines. The author firmly believes that the design of an algorithm does not end with its presentation in an algebraic language, but that it must change in detail as it is implemented on various machines.

REFERENCES

1. ROBERT T GREGORY
   *On the design of the arithmetic unit of a fixed word length computer from the standpoint of computational accuracy*

2. LEONARD J HARDING JR
   *Idiosyncracies of system 360 floating point*
   presented at SHARE XXVII Toronto Canada 1966

3. R W HAMMING W KAHAN H KUKI C L LAWSON
   *Panel discussion on machine implementation of numerical algorithms*
   Presented at SICNUM meeting Los Angeles California 1966

4. R MONTEROSSO
   *Differenze nella rappresentazione e nella arithmetica floating tra le serie IBM 700 7000 E la serie 360 errori di arrotondamento nelle operazioni elementari floating point dei calcolatori della serie IBM 360*
   Report Eur 2777 i Comunita Europea Del'Energia Atomica Ispra Italy April 1966

5. JOHN H WELSCH
   *Proposed floating point feature for IBM system 360*
   presented at SHARE XXVII Toronto Canada 1966

6. H R J GROSCH
   *Programmers: The industry's Cosa Nostra*
   Datamation 12 p 202 1966

7. RICHARD KING
   *Reduction of Roundoff Error in Gill Runge Kutta Integration*
   Argonne National Laboratory Applied Mathematics Division
   Technical Memorandum No 22 Argonne Illinois June 1961