ARA Journal, Volume 2005, No. 30

Impact of Floating-Point Arithmetic on Engineering Numerical Analysis

Farooq Saeed and Ahmad Z. Al-Garni
Aerospace Engineering Department, King Fahd University of Petroleum and Mineral
Mail Box 1637, Dhahran 31261, Saudi Arabia
farooqs@kfupm.edu.sa

Abstract: Floating-point number systems inevitably incur errors in representing both the input and the results of computation. Such errors can be minimized by careful selection and implementation of computational algorithms. Nevertheless, some problems are inherently sensitive (ill-conditioned) in that their solutions can change dramatically as a result of a small perturbation in their input, regardless of the algorithm used. On the other hand, poorly conceived algorithms can be numerically unstable in solving perfectly well conditioned problems. In backward error analysis, the accuracy of an approximate solution is assessed by determining a modified problem whose exact solution is the one actually obtained for the original problem. The objective of the paper is to review the floating-point number system and discuss its implications on the results of computations typical of those found in engineering numerical analysis.

Introduction to Floating-Point Number System

A floating-point number system [1-4] is characterized by four integers, namely: the base β, the precision t, i.e., the number of digits to be carried, and the exponent range [L, U] characterized by a lower limit L and an upper limit U. Thus, any number x in the floating-point system is represented as:

$$x = \pm\left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t}\right)\beta^e$$

where $0 \le d_1 \le \beta - 1, i = 1, \ldots, t$ and $L \le e \le U$. The part within the parenthesis is called the *fraction* or *mantissa*, and e is called the *exponent* of the floating-point number x. For example, for a number system such that $\beta = 10$, $t = 1$, $L = -1$, and $U = 1$, there are a total of 55 numbers in this system, i.e.:

$$x = 0, \pm\left(\frac{1}{10}\right)10^{-1}, \pm\left(\frac{2}{10}\right)10^{-1}, \ldots, \pm\left(\frac{9}{10}\right)10^{-1}, \pm\left(\frac{1}{10}\right)10^{0},$$
$$\pm\left(\frac{2}{10}\right)10^{0}, \ldots, \pm\left(\frac{9}{10}\right)10^{0}, \pm\left(\frac{1}{10}\right)10^{1}, \pm\left(\frac{2}{10}\right)10^{1},$$
$$\ldots, \pm\left(\frac{9}{10}\right)10^{1} \pm\left(\frac{1}{10}\right)10^{1}, \pm\left(\frac{2}{10}\right)10^{1}, \ldots, \pm\left(\frac{9}{10}\right)10^{1}$$
$$\Rightarrow x = 0, \pm 0.01, \pm 0.02, \ldots, \pm 0.09, \pm 0.1, \pm 0.2, \ldots,$$
$$\pm 0.9, \pm 1, \pm 2, \ldots, \pm 9 \Rightarrow \text{a total of } 55 \text{ numbers}$$

In other words, the floating point number system does not contain all possible numbers within its range but only a finite number. Thus, some numbers may not be defined within this finite precision. Table 1 lists some of the typical examples of floating-point number systems.

Table 1: Some floating-point number systems [3]

Machine	β, t, L, U
IEEE std. single precision,	2 (binary), 24, -125, 128
IEEE std. double precision,	2 (binary), 53, -1021,1024
Cray,	2 (binary), 48, -16384, 16383
IBM 360 single precision,	16 (hexadecimal), 6, -64, 63
IBM 360 double precision,	16, 14, -64, 63

Properties of Floating-Point Number Systems

- A floating-point number system is finite and discrete. Specifically, the number of floating-point numbers is given by: $2(\beta - 1)\beta^{t-1}(U - L + 1) + 1$.
- There is largest floating-point number and the *overflow* level associated with it is = $OFL \approx \beta^U$
- There is smallest positive floating-point number and the *underflow* level associated with it is = $UFL \approx \beta^L$
- Floating-point numbers are not uniformly distributed throughout their range, but are equally spaced only between successive powers of β.
- Not all real numbers are exactly represented in a floating-point system. [Note: In this paper, the floating-point approximation of a given real number x is denoted by $fl(x)$].

Properties of Floating-Point Arithmetic

- Some familiar laws of real arithmetic are not necessarily valid in a floating-point system. For example, floating-point addition and multiplication are commutative but not associative, i.e.:
$$(a + b) + c \neq (a + c) + b$$
- A real arithmetic operation on two floating-point numbers does not necessarily result in another floating-point number.
- If a number, that is not exactly represent able as a floating-point number, is entered into the computer, or is produced by subsequent arithmetic operations, then it must be rounded or chopped to the nearest floating-point number.
- Since, floating-point numbers are not equally spaced; the error made in such an approximation is not uniform.
- Ideally, $x \; flop \; y = fl(x \; op \; y)$, and most computers come close to this ideal as long as $x \; op \; y$ is with in the range of the floating-point system.

The accuracy of floating-point arithmetic depends upon the precision of the machine which is characterized by *machine epsilon*, also known as the unit round-off error, which is defined to be the smallest floating-point number ε_{mach} such that:

$$1 + \varepsilon_{mach} > 1$$

in floating-point arithmetic.

With rounded arithmetic: $\varepsilon_{mach} \approx \frac{1}{2}\beta^{1-t}$

With chopped arithmetic: $\varepsilon_{mach} \approx \beta^{1-t}$

In other words, machine epsilon determines the maximum relative error in representing a real number x in a floating-point system, i.e.:

$$\left|\frac{x - fl(x)}{x}\right| \le \varepsilon_{mach}$$

ε_{mach} is determined by the number of bits in the mantissa of a floating-point system, while *OFL* and *UFL* are determined by the number of bits in the exponent.

$$0 < UFL < \varepsilon_{mach} < OFL$$

For example, for base β = 10, precision t = 4, L = -10 to U = 10, the lowest floating-point number that will not perturb 1.0 since 1.0000000000 + 0.0000000001 = 1.0000000001. Here the underlined numbers indicate no change within the precision t (number of digits after the decimal point). Machine epsilon, e.g. for 5-digit base 2 number system is:

$$\varepsilon_{mach} = (0.00001)_2 = (0\times 2^{-1} + 0\times 2^{-2} + 0\times 2^{-3} + 0\times 2^{-4} + 1\times 2^{-5}) = 1/2^5 = 1/32$$

Example: Machine's *UFL*

The following program determines the machine's *UFL* which is typically 1×10^{-38}.

```
      EPS = 1.0
      WRITE(*,*) EPS
   10 EPS = EPS/2.0
      WRITE(*,*) EPS
      IF(EPS.GT.0.0) GOTO 10
```

Example: Round-off error

Consider a floating-point system $a \times 10^b$ where a (mantissa) has four and b (exponent) has two decimal digits, e.g., $4.236 \times 10^{00} = 4.236$, etc. In this system, the sum:

$$4.236 \times 10^{00} + 5.629 \times 10^{-04} = 4.236 + 0.0005629 = 4.2365629$$

must be reduced to four digits by either chopping (which yields 4.236×10^{00}) or rounding (which yields 4.237×10^{00} – the more accurate result). In either case, the difference between the true value and the floating-point value is called **round-off error**.

Example: Machine epsilon ε_{mach}

The following program demonstrates that rounding off to zero occurs when the number becomes smaller than the machine epsilon ε_{mach} which is 1×10^{-7}. (Note: ε_{mach} is the smallest number to perturb 1.0)

```
      EPS = 1.0
      WRITE(*,*) EPS
   10 EPS = EPS/2.0
      WRITE(*,*) EPS
      EPSP1 = EPS + 1.0
      IF(EPSP1 .GT. 1.0) GO TO 10
```

Absolute and Relative Errors [4]

Absolute error = True value – Approximate value

Relative error = (Absolute error) / (True value)

Example:

$$fl(x) = x(1+\delta), |\delta| \le \varepsilon_{mach},$$

or

$$\left|\frac{x - fl(x)}{x}\right| = |\delta| \le \varepsilon_{mach}$$

which means that the floating-point representation of a real number within the range of the floating-point system has a relative error of the order of machine epsilon. Ideally, for all floating-point operations:

$$x \; flop \; y = (x \; op \; y)(1 + \delta), \text{ with } |\delta| \le \varepsilon_{mach}$$

Example: Evaluating a Function

Consider the function $f(x)$, then absolute error is $f(x+h) - f(x) \approx hf'(x)$, whereas the relative error is

$$\frac{f(x+h) - f(x)}{f(x)} \approx h\frac{f'(x)}{f(x)}$$

For example, if $f(x) = e^x$, then the absolute error is $\approx he^x$, and the relative error $\approx h$. A stable algorithm is the one in which the computed solution exactly solves a perturbed problem (i.e. if coefficients are modified slightly). A well-conditioned problem is that for which the computed solution and the true solution are similar or the computed solution is a good approximation of true solution.

Sources of Error in Scientific Computing

- Hardware malfunction. Extremely rare today.
- Blunders (programmer mistakes). Unfortunately, not extremely rare.
- Experimental error. Due to limited precision of measuring instruments, laboratory equipment, sample sizes, etc.
- Modeling error. Due to omission of some physical features of the problem or system under study (e.g. friction, viscosity).
- Truncation error. Due to omission of some mathematical features of the mathematical model under study. (Linearized)
- Rounding error. Due to inexactness of floating-point numbers and arithmetic in representing real numbers and arithmetic.

Errors in the final results of a computation may reflect a combination of any or all of these sources, and may be amplified or magnified by the sensitivity or conditioning of the problem being solved and the stability of the algorithm being used.

Cancellation Errors and Loss of Significance

In floating-point arithmetic, significant digits are lost when adding floating-point numbers of greatly differing magnitudes. The reason being that if the exponents of the two numbers differ then the mantissa are shifted to match the exponent and then perform the operations of addition or subtraction. This could also be as a result of: (1) Overflow limit reached, (2)

Increment is smaller than , and (3) the relative error is less than ε_{mach}.

Example: The infinite series

$$\sum_{n=1}^{\infty} \frac{1}{n}$$

has a finite sum in floating-point arithmetic even though the real series is divergent. Similarly, significant digits are lost when subtracting numbers with nearly equal magnitudes.

Example: Effect of Double Precision

Let's compute the value of π using the following relation [1]: $p_{n+1} = 2^n \sqrt{2(1-\sqrt{1-(p_n/2^n)^2})}, p_2 = 2\sqrt{2}$

The following program demonstrates that double precision changes the machine epsilon but not the *UFL*.

```
      P(2) = 2.0*SQRT(2.0)
      WRITE(*,*) P(2)
      DO 30 N = 3, 60
         A    = N
         B    = 2.0**(A-1)
         C    = (P(N-1)/B)**2.0
         D    = 2.0*(1.0-SQRT(1.0-C))
         P(N) = B*SQRT(D)
         WRITE(*,*) N,B,C,D,P(N)
   30 CONTINUE
```

Output:

n	p_n
2	0.28284271E+01
3	0.30614675E+01
4	0.31214452E+01
…	…
26	0.31622777E+01
27	0.31622777E+01
28	0.34641016E+01
29	0.40000000E+01
30	0.00000000E+00
31	0.00000000E+00

Example: Summing the Taylor's series for e^x

We know that the Taylor's series $e^x = 1 + x + x/2! + \cdots$ converges for all values of x. However, disastrous results are obtained for $x < 0$, on a machine (VAX) with $\varepsilon_{mach} \approx 10^{-7}$. The following is the FORTRAN implementation of the series summation.

```
      WRITE (*,*) 'ENTER X'
      READ (*,*) X
      SUM  = 1.0
      TERM = 1.0
      I    = 1
  1   TERM = TERM*(X/I)
      IF (SUM+TERM .EQ. SUM) THEN
        WRITE (*,*) 'E(X) = ', SUM, EXP(X)
C  Stop summing when there is no contribution
C  to the series
      ELSE
        SUM = SUM + TERM
        I   = I + 1
        GO TO 1
      ENDIF
      END
```

Table 2 compares the exact solution with the results computed on VAX and CDC ($\varepsilon_{mach} \approx 10^{-14}$) machines. Although the results are improved with a smaller ε_{mach}, the results show errors for $x < -10$.

Table 2: Taylor's series summation for e^x

x	Exact value of e^x	VAX	CDC
1	2.718282	2.718282	2.718282
5	148.4132	148.4132	148.4132
10	22026.46	22026.47	22026.47
15	3269017.	3269017.	3269017.
20	4.8516520×10^{8}	4.8516531×10^{8}	4.8516520×10^{8}
-1	0.3678795	0.3678794	0.3678794
-5	6.7379470×10^{-3}	6.7377836×10^{-3}	6.7379470×10^{-3}
-10	4.5399930×10^{-5}	-1.6408609×10^{-4}	4.5399952×10^{-5}
-15	3.0590232×10^{-7}	-2.2377001×10^{-2}	3.0508183×10^{-7}
-20	2.0611537×10^{-9}	1.202966	3.865358×10^{-7}

Example: Quadratic Formula

Cancellation error and other numerical difficulties need not involve a long series of computations. For example, use of the standard formula for the roots of a quadratic equation is fraught with numerical pitfalls. As every school child learns, the solutions of the quadratic equation

$$ax^2 + bx + c = 0$$

are given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For some values of the coefficients, naive use of this formula in floating-point arithmetic can suffer overflow, underflow, loss of significance, or catastrophic cancellation. ***Hence for math computations, the order of adding terms should be from lowest in magnitude to highest for max accuracy***.

Example: Finite Difference Approximation of Derivative

$$f'(x) \approx \frac{f(x+h) + f(x)}{h}$$

Here we would like to have h small so that the approximation is accurate, but if h is too small, then $fl(x + h)$ may not differ from $fl(x)$. Even if $fl(x+h) \neq fl(x)$, there is a loss of significance in computing the difference $f(x+h) - f(x)$. We might even have $fl(f(x+h)) = fl(f(x))$. Thus, there is a trade off between the truncation error and rounding error in choosing the size of h. A rule of thumb is that it is usually best to perturb about half the digits of x by taking $h \approx \sqrt{\varepsilon_{mach}}\,|x|$

The rounding error can be reduced by working with higher precision arithmetic. Truncation error can be reduced by using a more accurate formula, such as the centered difference approximation

$$f'(x) \approx \frac{f(x+h) + f(x-h)}{2h}$$

Example: Computing Residuals

Problem: Solve $ax = b$

Let z be approximate solution. Here the residual is:

$$r = b - a\,z$$

In floating-point arithmetic,

$$a \times fl\ z = a\ z\ (1+\delta_1)$$

So

$$b - fl\ (\mathrm{a} \times fl(z)) = [b - az(1+\delta_1)](1+\delta_2)$$
$$= [r - az](1+\delta_2) = r + \delta_2 r - \delta_1 az - \delta_2\delta_1$$
$$\approx r\ + \delta_2 r - \delta_1 b - \delta_1\delta_2 b$$

But $\delta_1 b$ may be as large as $\varepsilon_{mach} b$, which may be as large as r. Thus, only higher precision will enable a meaningful computation of the residual r.

Example: System of two linear equations

$$0.780x + 0.563y = 0.217$$
$$0.457x + 0.330y = 0.127$$

In a 3-digit decimal arithmetic, the solution is

$$x = 1.71, \quad y = -1.98$$

Substituting these numbers back in the equations gives

$$0.780(1.71) + 0.563(-1.98) - 0.217 = 0.00206$$
$$0.457(1.71) + 0.330(-1.98) - 0.127 = 0.00107$$

The result, called the residual, is small considering the 3-digit decimal arithmetic. However, the exact solution to the system is:

$$x_{\text{exact}} = 1.000, \quad y_{\text{exact}} = -1.000$$

Although this example is very simple in nature, it does illustrate that the computed solution can be very different from the exact based on the floating-point system being used.

Backward Error Analysis

Analyzing the forward propagation of rounding errors is very difficult due to the failure of floating-point arithmetic to satisfy the usual laws of real arithmetic. It also tends to produce very pessimistic results, due to "worst case" assumptions at every stage. An alternative approach, developed primarily by James Wilkinson [1], is backward error analysis. Consider the approximate solution obtained to be the exact solution for a modified problem. Then ask how large a modification to the original problem is required to give the result actually obtained. In terms of backward error analysis, a problem is well conditioned and an algorithm for solving it is stable, if the approximate solution obtained is the exact solution to a "nearby" problem. The great practical advantage of backward analysis is that real arithmetic (with its associativity, etc.) can be used in the analysis.

Sensitivity and Conditioning

Numerical difficulties are not always due to an ill conceived formula or algorithm, but may be inherent in the problem being solved. For example, consider the problem of computing values of the cosine function for arguments near $\pi/2$. Let $x = \pi/2$ and let h be a small perturbation to x. Then the error in computing $\cos(x+h)$ is given by:

$$\text{Absolute error} = \cos(x+h) - \cos(x) = -h\sin(x) = -h$$

And hence,

$$\text{Relative error} = -h\tan(x) \approx \infty$$

Thus, small changes in x near $\pi/2$ cause large relative changes in $\cos(x)$ regardless of the method for computing it. For example,

$$\cos(1.57079) = 0.63267949 \times 10^{-5}$$
$$\cos(1.57078) = 1.63267949 \times 10^{-5}$$

This problem is serious and not just a pathological example, since in root finding problems one is interested precisely in those points where $f(x) = 0$.

Conclusions

In this paper, some of the important features of a floating-point arithmetic are presented. Although the material available in literature is very extensive in this regards, the work presented here will attract the reader in a more thorough study of the subject. As stated earlier, floating-point number systems inevitably incur errors in representing both the input and the results of computation. Such errors can be minimized by careful selection and implementation of computational algorithms. Nevertheless, some problems are inherently sensitive (ill-conditioned) in that their solutions can change dramatically as a result of a small perturbation in their input, regardless of the algorithm used. On the other hand, poorly conceived algorithms can be numerically unstable in solving perfectly well conditioned problems.

Techniques such as backward error analysis can be used to assess the accuracy of an approximate solution by determining a modified problem whose exact solution is the one actually obtained for the original problem. The implications of using a certain algorithm in the computations typical of those found in engineering numerical analysis should be carefully analyzed at each step before the results are inferred.

Acknowledgements

The authors are also grateful to King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for its support in accomplishing this study.

References

1. Wilkinson, J.H., *Rounding Errors in Algebraic Processes*, Dover Publications, Reprint edition, May 1994.
2. Kahaner, D., Moler, C., and Nash, S., *Numerical Methods and Software*, Prentice Hall, November 1988.
3. IEEE Standard for Binary Floating Point Arithmetic, Institute of Electrical & Electronics Engineering, March 1985.
4. Roy, M. R., *A History of Computing Technology*, Wiley-IEEE Computer Society Pr; 2nd edition, March 1997.
5. Abramowitz, M., and Stegun, I. A., *Handbook of Mathematical Functions*, Dover Publications, June 1974.