Computer System Support for Scientific and Engineering Computation
Lecture 1 - May 3, 1988

1 Mysteries of Floating Point Arithmetic

This chapter serves to explain a certain state of mind where people too commonly perceive floating point arithmetic as a mysterious and irrational subject. It is pertinent to understand this all-pervasive state of mind; only then can we go on to understand why floating point is capable of being mathematically manageable, provided we take care of its design. A designer must understand that he is working in a mathematical domain in which taking shortcuts is simply not a good practice as it may violate reasonable expectations based on mathematical principles. What may seem irrelevant or unimportant to him may be very important to someone else. Whatever corners we cut today may affect others in the future just as whatever inconsistencies we experience today are the deeds of past culprits. Facts and examples indicate that perplexing things, some of which are intrinsic and easily explained, happen when care has been neglected in designs. These puzzling occurrences are arbitrary and avoidable. Investing resources such as time and money in careful designs today will eventually accrue benefits to others and is more cost effective in the long run. Contrary to other norms of life, when we do our jobs right in floating point arithmetic others will not know!

There are certain perculiarities about floating point arithmetic which can be frightening and mysterious to people who are not familiar with it, as well as to people who deal with it every day. In floating point arithmetic, unlike in most other computing environments, you cannot always rely on what you see and you may not always obtain what you anticipate. Listed below are some very frigthening facts about floating point arithmetic :

1. What you see is often not what you have.
2. What you have is sometimes not what you wanted.
3. If what you have hurt you, you will probably never know how or why.
4. Things go wrong too rarely to be properly appreciated, but not rarely enough to be ignored.
5. Items 1 to 4 do not constitute carte blanche to build floating point any way you like.

Let us look at a few examples which illustrate and substantiate these facts.

Shing Ma
11 May 88

1.1 What You Get is Not What you Expected

Consider the following program :

```
           ----------- Borland  Turbo Basic  on an  IBM PC -----------

       +---------------------------- Edit ----------------------------+
       |                                                              |
       |   CLS                                                        |
       |   p = 2 :  for i=1 to 6 :  p = p*p :  next i                 |
       |   pp1 = p + 1 :   pm1 = p - 1                                |
       |   d = pp1 - pm1                                              |
       |   Print "  We expect d = 2 ,  but actually  d = "; d ; " ,"  |
       |   Print "  although  (p+1) - (p-1) = "; (p+1) - (p-1) ; " . !"|
       |   Print                                                      |
       |   Print "  What you get isn't necessarily what you expected." |
       |   End                                                        |
       +--------------------------------------------------------------+
```

According to the laws of algebra one expects both d and $(p+1)-(p-1)$ to equal 2. However, what one gets is $d=0$ and $(p+1)-(p-1)=1$ as shown below :

```
       +---------------------------- Run -----------------------------+
       |                                                              |
       |     We expect d = 2 ,  but actually  d =  0  ,               |
       |     although  (p+1) - (p-1) =  1  . !                        |
       |                                                              |
       |     What you get isn't necessarily what you expected.        |
       |                                                              |
       +------------------+----------------------+--------------------|

                 WHAT YOU GET ISN'T NECESSARILY WHAT YOU EXPECTED.
```

It is evident that we are not getting what is expected as the laws of algebra appear to have been violated.

What seems mysterious, incorrect and irrational in the above program can be easily explained. If the value of p is a large enough power of 2, $p+1$ and $p-1$ are actually rounded to p, thereby resulting in $d=0$. In some computers, arithmetic operations are performed on arithmetic registers or stack which have greater precision than those in which the variables are declared. If the value of p is just large that $p+1$ rounds to p but $p-1$ can be represented distinctly from p, then $(p+1)-(p-1)$ yields 1. So, what seems strange and illogical at first can actually be explained by understanding the representation of floating point numbers and the nature of floating point arithmetic operations.

1.2 What You See is Not Necessarily What You Get

Here is another simple illustration in which the result does not match the expectation as things are not always what we perceive them to be. Consider the following program :

$q = 3.0/7.0$
if $q = 3.0/7.0$ then A
else B

One would expect A to be executed but B was executed instead. The following program, which prints the values of q and 3.0/7.0, provides an explanation for the bizarre and unexpected result.

```
------------ Borland  Turbo Basic  on an  IBM PC ------------

+------------------------ Edit ------------------------------+
|     D:WYSINWYG.BAS                                          |
|                                                             |
|     CLS                                                     |
|     q = 3.0/7.0                                             |
|     Print "  The value of  q = "; q                         |
|     Print "     but  3.0/7.0 = "; 3.0/7.0                   |
|     Print                                                   |
|     Print "  What You See Is Not Necessarily What You Get." |
|     End                                                     |
+-------------------------------------------------------------+
+------------------------ Run -------------------------------+
|                                                             |
|     The value of  q =  .4285714328289032                     |
|        but  3.0/7.0 =  .4285714285714286                    |
|                                                             |
|     What You See Is Not Necessarily What You Get.           |
|                                                             |
+----------------+-----------------------------+--------------+
```

The values of q and 3.0/7.0 are different since 3.0/7.0 are computed but rounded to different precisions in each case. In fact, both q and 3.0/7.0 are approximations of the actual value of 3.0/7.0 which cannot be represented in finite precision. Once again, we see that floating point arithmetic is capable of producing counter-intuitive results which are actually not as illogical as they appear. People responsible for compilers can minimize much fear and panic by documenting such phenomena.

1.3 How Often Do Errors Occur and Can We Ignore Them?

When errors, which occur rarely, surface as minute imperfections and seem incomprehensible, there is a tendency and willingness to ignore them. Such a compromise introduces an element of risk. In numerical computations, what failure rate is reasonable and tolerable? Engineers use concepts such as probability of failure, MTBF (mean time between failure), MTTR (mean time to repair) and confidence limits to quantify the occurrences of events. For instance, in the automobile or the telephone industries a failure rate of 1 in 1 million is probably reasonable and acceptable. However such an error in numerical computations is unacceptable and intolerable. In fact the failure rate of 1 in 1 billion is of significant magnitude and is intolerable. A failure rate of 1 in 1 billion for a computer which operates at 10 MFLOPS or 10^7 flops per second averages to the occurrence of an error every 10^5 seconds (approximately 30 cpu hours). If an engineer uses 3 cpu hours a day, one can expect an error to surface every 2 weeks. If it takes an engineer a week to detect and debug such an error, a significant amount of time is spent on debugging instead of performing other useful tasks, which is a loss of valuable personnel resources.

1.4 Nightmares for Programmers

Some numerical algorithms are stable in theory but in practice they may fail on some computers under certain circumstances. Take the following singular value decomposition program for instance :

$$\begin{array}{lll} & \vdots & \\ (1) & \lambda = 1.0 - f/h & (0 \le f \le h) \\ (2) & \mu = \ldots & \\ (3) & \rho = \sqrt{\lambda^2 + \mu^2} & \\ & \vdots & \\ (4) & \ldots = \mu/(\rho + \lambda) + \ldots & (\mu \ne 0) \end{array}$$

By virtue of branches, $0 \le f \le h$ in (1) and $\mu \ne 0$ in (4). Since $\rho \ge |\mu|$ and $|\mu| > 0$, evaluating $\frac{\mu}{(\rho+\lambda)}$ should be safe as $\frac{\mu}{(\rho+\lambda)} \le 1.0$.

There is an unpredictable feature of the CRAY where $\frac{f}{h} \le 1.0$ is not guaranteed even though $0 \le f \le h$. When $\frac{f}{h} > 1.0$, which implies that $\lambda < 0.0$, the evaluation of $\frac{\mu}{(\rho+\lambda)}$ may malfunction because when μ is very tiny, we have $\rho \approx |\lambda|$ which will result in a divide by zero in (4). As we cannot be assured that $\frac{f}{h} \le 1.0$ on a CRAY, we may need to test for $\lambda \ne -\rho$ before entering (4) to avoid a divide by zero. It defies mathematical rules for $\frac{f}{h} > 1.0$, but this embarrassing situation is not an impossibility for the CRAY due to the way division and multiplication are performed.

- A divider is an inherently complicated device and it requires quite a large area on a chip. Since the number of occurrences of divides in most algorithms are relatively low, there is a tendency to dedicate as little resources as possible to develop it.

  One of the ways in which CRAY implements its division is by "division without a divider". Division on a CRAY is based on Newton's iteration. Given A and B, $Q = \frac{A}{B}$ can be computed as follows :

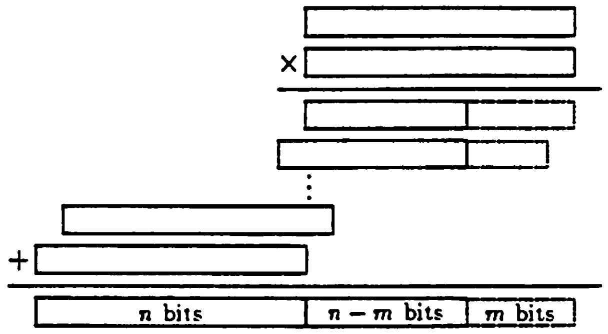
  1. Get $r := 1/B$ approximately (by table look-up)
  2. $R = r \cdot (2 - B \cdot r) \cdots$ perhaps repeated
  3. $Q = A \cdot R$

  According to Euler's formula, if $|x| < 1$, then

  $$\frac{1}{1-x} = (1+x)(1+x^2)(1+x^4)(1+x^8)\cdots$$

  Euler's formula implies that at the cost of 2 multiplications, one in squaring and the other in multiplying the factor, one can increase the number of correct digits by a factor of 2. The value of r is obtained by table look-up. This process involves 2 abbreviated multiplications whose total cost amounts to about 1 multiplication time. Computing $R = r \cdot (2 - B \cdot r)$ and $Q = A \cdot R$ requires 2 and 1 multiplications, respectively. The expression $2 - B \cdot r$ actually requires just 1 multiplication time since it is merely the two's complement of $B \cdot r$. This implementation of division on a CRAY, therefore, takes approximately 4 multiplication times, but if two multiplication units are available only 3 multiplication times are needed. (Can you restructure the algorithm to accomplish this?) This implementation of CRAY's division is heavily dependent on the way it handles multiplications. In other words, if its multiplications are inaccurate then its divisions are inaccurate too.

- A problem with CRAY's implementation of division is due to the manner in which it performs its multiplications. The following diagram illustrates how we can multiply two n bit numbers :



The result is a $2n$ bit number which is then rounded to a n bit number, whereby the bottom n bits are dropped after rounding. Since the bottom n bits are dropped eventually, CRAY does not compute the lowest m ($m < n$) bits to speed up the multiplication. Since the last m bits are ignored, the carrys that they should cause into the leading bits do not occur, which may therefore affect the final result in the rounding process. CRAY has, therefore, attained speed of its multiplications by sacrificing accuracy.

As CRAY's multiplication is potentially inaccurate, its division, which is heavily dependent on its multiplication, is also potentially inaccurate. These discrepancies will eventually return to haunt us. An excellent example is that on a CRAY we cannot be certain of $\frac{f}{h} \leq 1.0$, given that $0 \leq f \leq h$. Because we cannot rely on CARY's arithmetic to fully conform with the laws of algebra, algorithms which have been theoretically proven to be stable may malfunction in practice.

1.5 Underflow and Overflow are Not Uncommon

There is another vulnerability in the way in which CRAY performs its division since it computes the reciprocal first. The possibility exists, though it may be rare, that when we divide a tiny number by another tiny number, intending to obtain a reasonable quotient, it is possible that when the divisor is close to the underflow threshold, its reciprocal may overflow. This means that it is possible for a perfectly legal division to fail when computing the reciprocal on the CRAY.

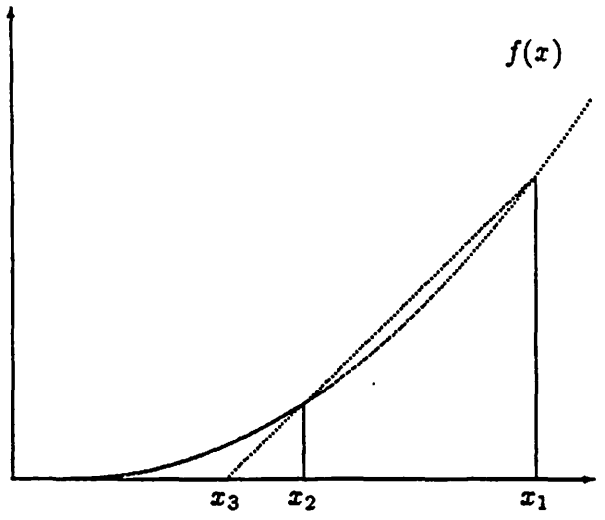
Numbers close to the underflow or overflow threshold are not at all uncommon. As an example, consider a linear control theory problem where we compute $det(A(x))$, the determinant of matrix $A(x)$. The values of x when $det(A(x)) = 0$ correspond to the resonant frequencies, possibly complex numbers, which are related to stability properties. Matrix A is usually large and sparse; in practice, it is not unreasonable for A to be 10000×10000. One way to determine $det(A(x))$ is by performing some form of triangular factorization on $A(x)$, such as Gaussian elimination. Let $B(x)$ be the upper triangular matrix obtained from factorizing $A(x)$. The value of $det(A(x))$ is the product of the diagonal elements of $B(x)$.

Let us assume that $A(x)$ is a 10000×10000 matrix. If all the diagonal elements of $B(x)$ are 0.7, $det(A(x)) = 0.7^{10000}$ which may cause an underflow. To circumvent the underflow problem, we may scale each element of $A(x)$ by multiplying it by 2. Scaling by 2 on a binary computer will not result in any rounding errors. By scaling A by 2 we obtain $det(A(x)) = 1.4^{10000}$ which may overflow. It is, therefore, not uncommon in practice to encounter an overflow when we scale a problem to avoid underflow, or vice-versa.

When one encounters the underflow situation in the linear control problem, there is a tendency to assume that $det(A(x)) = 0$ and that the resonant frequency has been determined. The resonant frequency thus determined is wrong, but the engineer may not be aware of it. This situation is especially apparent on computers with no hardware to detect underflow. Programmers sometimes solve the scaling problem by scaling after the elimination of a row or a column, but in doing so the code is no longer vectorizable.

1.6 Yet Another Programmer's Nightmare

The eigenvalues of a matrix A are the roots of its characteristic equation. Some eigenvalue solvers compute the eigenvalues by determining the roots of $f(x) = det(A(x)) = det(A-xI)$. One of the methods to compute the roots of a function $f(x)$ is the secant method illustrated below :



Using initial guesses, x_1 and x_2, we compute the next approximation to the root by

$$x_3 = x_2 + \frac{(x_1 - x_2)}{1 - \frac{f(x_1)}{f(x_2)}}$$

If the values of $f(x_1) = det(A(x_1))$ and $f(x_2) = det(A(x_2))$ are very small or very large, it is conceivable that x_3 is a reasonable number. To obtain x_3, $\frac{f(x_1)}{f(x_2)}$ is required, but on some machines like a CRAY, this may not be computable. Once again, this is due to the need to compute the reciprocal of $f(x_2)$. Here we have an expression which seems numerically reasonable but may malfunction due to the way certain arithmetic operations are implemented.

1.7 A Funny Fact About Divide

On some computers divisions are not properly rounded, but most people are not aware of it since we do not always use the properties of divide. There are various properties of divide, such as $\frac{f}{h} \leq 1.0$ if $0 \leq f \leq h$ and $i = \frac{i}{d} \times d$, but which properties should we honor? A problem with floating point arithmetic is that if we do not honor some of their properties, it is very likely that there are programs which will fail because they depend on them. The short program below is an excellent example which exhibits that if division is not rounded properly, as in the IEEE 754 standard, it will terminate prematurely :

```
for $i = 1, 2, 3, \cdots, 8000000$ in turn do
    for $d = 2, 3, 4, 5, 6, 8, 9, 10, 12, \cdots, 32768$ in turn do      ← No primes!
        real $Q = i/d$      (rounded)                                [illegible]
        real $X = Q \cdot d$      (rounded)
        if $X \neq i$ exactly then
            SHOUT "$X \neq i$ !"
            STOP
        endif
    next $d$
next $i$
OBSERVE "$X = i$ always!"
```

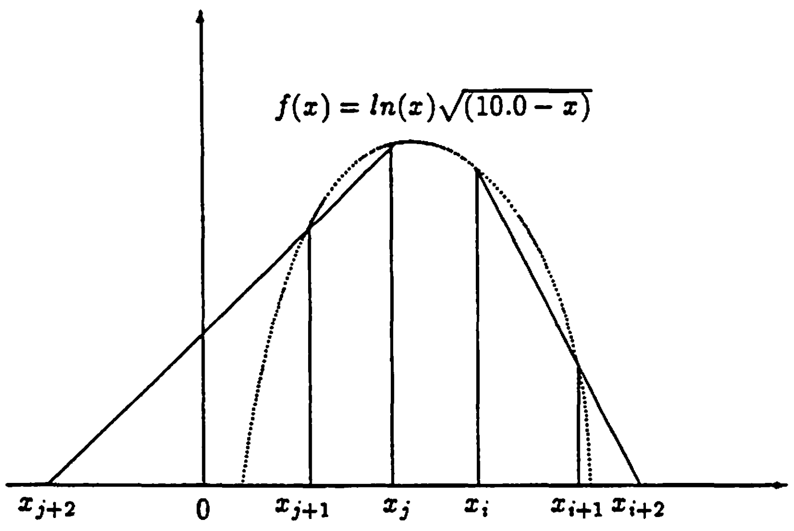
When we divide an integer by an integer as in $Q = \frac{i}{d}$, we generally obtain a floating point number which has been rounded. When we multiply a floating point number by an integer as in $X = Q \cdot d$ we expect a floating point number which has been rounded. According to the laws of algebra X should be an integer. On computers whose arithmetic conforms to the IEEE 754 standard, $X = i$ for all d. However, for computers whose arithmetic do not conform with the standard, $X \neq i$ for some j, which violates some mathematical rules. An important point here is that if this short program can detect violation of some mathematical rule, then there are certainly application programs which will fail because of improper rounding.

2 Exception Handling

Exception handling is a subject that is full of prejudice and other predispositions. For instance, in school one is taught that division by zero is a no-no and he who does it must be punished. This trend of thought persists in the world of numerical computations for when there is a division by zero, many machines suspend computation. A proper way to treat division by zero, and square root of a negative number is to treat them as exceptions and not errors. In the IEEE 754 standard, a NaN (not a number) is returned as the result of many exceptional operations.

In order for software to be truly portable with only the need for recompilation, exceptions must be handled appropriately. Exceptions do not imply errors in computation but merely our inability to cope with certain situations. Underflow, overflow, division by zero and inexactness are some examples of exceptions. Exceptions are errors only if they are handled improperly. A problem with exception handling is the adoption of any single policy, like suspension of a program when a division by zero is encountered. The essence of exceptions is that no policy adopted in advance can be found unexceptionable. There are always situations which warrant exceptions to the standard policy. That's why they're called exceptions.

Lets us consider an example where computation suspension due to an exception is inappropriate. In fact, this example shows that it is not irrational to resume execution upon encountering an exception. A method to obtain the roots of a function $f(x) = \ln(x) \cdot \sqrt{10.0 - x}$ called the secant iteration is illustrated below :



Let x_i and x_{i+1} be the initial guesses, and the next guess $x_{i+2} > 10.0$. Since $x > 10.0$ we have the square root of a negative number which would raise an exception. Similarly, if the initial two guesses are x_j and x_{j+1}, we have $x_{i+2} < 0.0$ but the natural log of a negative number also causes an exception. In the environment where one is punished for trying to take the square root ~~of~~ or the log of a negative number, the program will abort. If exceptions are handled with care, abortion of the program is not necessary; in fact, one can always obtain a solution to this problem. Whenever we have an exception, we know that our new guess is outside the domain. In the root finder subroutine, we can program it so that when the guess is outside the domain, we retract the guess and provide another guess which is hopefully within the domain. For instance, instead of x_{j+3}, which is outside the domain, we can use $\frac{x_{j+2}+x_{j+3}}{2}$ as the next guess instead. Hence, by handling exceptions appropriately, we will eventually obtain one of the roots of $f(x)$.

In subsequent chapters, we will explore and discuss exception handling in greater depth.