Scientific Computing: IEEE arithmetic

Jonathan Goodman

January 29, 1996

The IEEE floating point standard is a set of rules issued by the IEEE (Institute of Electrical and Electronics Engineers) on computer representation and processing of floating point numbers. Today, most computers claim to be IEEE compliant but many cut corners in what (they consider to be) minor details. The standard is currently being enlarged to specify some details left open in the original standard, mostly on how programmers interact with flags and traps. The standard has four main goals:

(1) To make floating point arithmetic as accurate as possible.

(2) To produce sensible outcomes in exceptional situations.

(3) To standardize floating point operations across computers.

(4) To give the programmer control over exception handling.

Point (1) is achieved in two ways. The standard specifies exactly how a floating point number should be represented in hardware. It demands that operations (addition, square root, etc.) should be as accurate as possible. For point (2), the standard introduces `inf` (infinite) to indicate that the result is larger than the largest floating point number, and `NaN` (Not a Number) to indicate that the result makes no sense. Before the standard, most computers would abort a program in such circumstances. Point (3) has several consequences: (i) that floating point numbers can be transferred from one IEEE compliant computer to another in binary without the loss of precision and extra storage associated with conversion to a decimal ASCII representation, (ii) that the details of floating point arithmetic should be understood by the programmer, and (iii) that the same program run on a different computer should produce exactly identical results, down to the last bit. The last one is not true in practice, yet.

The IEEE standard specifies exactly what floating point numbers are and how they are to be represented in hardware. The most basic unit of information that a computer stores is a *bit*, a variable whose value may be either 0 or 1. Bits are organized into groups of 8 called bytes. The most common unit of computer number information is the 4 byte (32 bit) word. For higher accuracy this is doubled to 8 bytes or 64 bits. There are 2 possible values for a bit, there are $2^8 = 256$ possible values for a byte, and there are 2^{32} different 4 byte words, about 4.3 billion. A typical computer should take less than an hour to list all 4.3 billion 32 bit words. To get an idea how fast your computer is, try running the program in figure 1. This will keep going as long as there are positive integers (half the 4 byte words represent positive integers). This program relies on the assumption that when the computer adds one to the largest integer, the result will be the smallest (most negative) integer. This is true in almost all computers.

```
#include <iostream.h>

main() {  // Count until you reach overflow.

   int increment = 1;
   int count     = 1;

   cout << "Start counting ... " << endl;

   while ( count > 0 ) {
      count = count + increment; }

   cout << "How long did it take to overflow?" << endl; } ;
```

Figure 1: A C++ program that counts until it overflows.

The two kinds of number are fixed point (integer) and floating point (real). A fixed point number has type `int` in C and type `integer` in FORTRAN. A floating point number has type `float` in C and `real` in FORTRAN. In most C compilers, a `float` by by default has 8 bytes instead of 4.

Integer arithmetic is very simple. There are $2^{23} \approx 4 \times 10^9$ 32 bit integers filling the range from about $-2 \cdot 10^9$ to $2 \cdot 10^9$. Addition , subtraction, and multiplication are done exactly whenever the answer is within this range. Most computers will do something unpredictable when the answer is out of range (overflow). The disadvantages of integer arithmetic are both that it can not represent fractions and that it has a narrow range of values. The number of dollars in the US national debt cannot be represented as a 32 bit integer.

A floating point (or "real") number is a computer version of the exponential (or "scientific") notation used on calculator displays. Consider the example expression:

$$-.2491\mathrm{E} - 5$$

which is one way a calculator could display the number -2.491×10^{-6}. This expression consists of a sign bit, $s = -$, a mantissa, $m = 2491$ and an exponent, $e = -5$. The expression $s.m\mathrm{E}e$ corresponds to the number $s \cdot .m \cdot 10^e$.

The IEEE format replaces the base 10 with base 2, and makes a few other changes. When a 32 bit word is interpreted as a floating point number, the first bit is the sign bit, $s = \pm$. The next 8 bits form the "exponent", e, and the remaining 23 bits determine the "fraction", f. There are two possible signs, 256 possible exponents (ranging from 0 to 255), and $2^{23} \approx$ 8.4 million possible fractions. Normally a floating point number has the value

$$x = \pm 2^{e-127} \cdot (1.f)_2 \quad ,$$

where f is base 2 and the notation $(1.f)_2$ means that the expression $1.f$ is interpreted in base 2. Note that the mantissa is $1.f$ rather than just the fractional part, f. In base 2 any number (except 0) can be normalized so that the mantissa has the form $1.f$. There is no need to store the "1." explicitly. For example, the number $2.752 \cdot 10^3 = 2572$ can be written

$$\begin{aligned} 2752 &= 2^{11} + 2^9 + 2^7 + 2^6 \\ &= 2^{11} \cdot \left(1 + 2^{-2} + 2^{-4} + 2^{-5}\right) \\ &= 2^{11} \cdot (1 + (.01)_2 + (.0001) + (.00001)_2) \\ &= 2^{11} \cdot (1.01011)_2 \quad . \end{aligned}$$

Thus, the representation of this number would have sign $s = +$, exponent $e - 127 = 11$ so that $e = 138 = (10001010)_2$, and fraction $f = (01011000000000000000000)_2$. The entire 32 bit string corresponding to $2.752 \cdot 10^3$ then is:

$$\underbrace{1}_{s}\underbrace{10001010}_{e}\underbrace{01011000000000000000000}_{f} \quad .$$

The exceptional cases $e = 0$ (which would correspond to 2^{-127}) and $e = 255$ (which would correspond to 2^{128}) have complex and carefully engineered interpretations that make the IEEE standard distinctive. If $e = 0$, the value is

$$x = \pm 2^{-126} \cdot 0.f \quad .$$

This feature is called "gradual underflow". ("Underflow" is the situation in which the result of an operation is not zero but is closer to zero than any floating point number.) The corresponding numbers are called "denormalized". Gradual underflow has the consequence that two floating point numbers are equal, $x = y$, if and only if subtracting one from the other gives exactly zero.

The use of denormalized (or "subnormal") numbers makes sense when you consider the spacing between floating point numbers. If we exclude denormalized numbers then the smallest positive floating pont number (in single precision) is $a = 2^{-126}$ (corresponding to $e = 1$ and $f = 00\cdots00$ (23 zeros)) but the next positive floating point number larger than a is b, which also has $e = 1$ but now has $f = 00\cdots01$ (22 zeros and a 1). The distance between b and a is 2^{23} times smaller than the distance between a and zero. That is, without gradual underflow, there is a large and unnecessary gap between 0 and the nearest nonzero floating point number.

The other extreme case, $e = 255$, has two subcases, `inf` (for infinity) if $f = 0$ and `NaN` (for Not a Number) if $f \neq 0$. Both C and FORTRAN print "`inf`" and "`NaN`" when you print out a variable in floating point format that has one of these values. The computer produces `inf` if the result of an operation is larger than the largest floating point number, in cases such as `x*x*x*x` when $x = 5.2 \cdot 10^{15}$, or `exp(x)` when $x = 204$, or `1/x` if $x = \pm 0$. (Actually $1/+0 = +$`inf` and $1/-0 =$ `-inf`). Other invalid operations such as `sqrt(-1.)`, `log(-4.)`, and `inf/inf`, produce `NaN`. It is planned that f will contain information about how or where in the program the `NaN` was created but this is not standardized yet. It might be worthwhile to look in the hardware or arithmetic manual of the computer you are using.

Exactly what happens when a floating point exception, the generation of an `inf` or a `NaN`, occurs is supposed to be under software control. There is supposed to be a flag (an internal computer status bit with values 0 or 1, something like the open/closed sign on a shop) that the program can set. If the flag is on, the exception will cause a program interrupt (the program will halt there and print an error message), otherwise, the computer will just give the `inf` or `NaN` as the result of the operation and continue. The latter is usually the default. In practice, it may be hard for the programmer to figure out how to set the arithmetic flags and other arithmetic defaults.

The floating point standard specifies that all arithmetic operations should be a accurate as possible within the constraints of finite precision arithmetic. The result of arithmetic operations is to be "the exact result correctly rounded". This means that you can get the computed result in two steps. First interpret the operand(s) as mathematical real number(s) and perform the mathematical operation exactly. This usually gives a result that cannot be represented exactly in IEEE format. The second step is to "round" this mathematical answer, that is, replace it with the IEEE number closest to it. Ties (two IEEE numbers the same distance from the mathematical answer) are broken in some way (e.g. round to nearest even). Any operation involving a `NaN` produces another `NaN`. Operations with `inf` are common sense: `inf` $+$ *finite* $=$ `inf`, `inf/inf` $=$ `NaN`, *finite*/`inf` $= 0.$, `inf` $-$ `inf` $=$ `NaN`.

From the above, it is clear that the accuracy of floating point operations is determined by the size of rounding error. This rounding error is determined by the distance between neighboring floating point numbers. Except for denormalized numbers, neighboring floating numbers differ by one bit in the last bit of the fraction, f. This is one part in $2^{23} \approx 10^{-7}$ in single precision. Note that this is relative error, rather than absolute error. If the result is on the order of 10^{12} then the roundoff error will be on the order of 10^5. This is sometimes expressed by saying that `z = x + y` produces $(x+y)(1+\epsilon)$ where $|\epsilon| \leq \epsilon_{\mathbf{mach}}$, and $\epsilon_{\mathbf{mach}}$, the "machine epsilon" is on the order of 10^{-7} in single precision.

A related remark about IEEE arithmetic is its scale invariance. The relative distance between neighboring floating point numbers does not depend too much on the size of the numbers, as long as they are not denormalized. A common definition of $\epsilon_{\mathbf{mach}}$ is that it is the smallest positive floating point number so that $1 + \epsilon_{\mathbf{mach}} \neq 1$. This is in keeping with a general principle in numerical computing. We should always measure error in relative terms rather than absolute terms.

Double precision IEEE arithmetic used 8 bytes (64 bits) rather than 4 bytes. There is one sign bit, 11 exponent bits, and 52 fraction bits. Therefore the double precision floating point precision is determined by $\epsilon_{\mathbf{mach}} = 2^{-52} \approx 5 \cdot 10^{-16}$. That is, double precision arithmetic gives roughly 15 decimal digits of accuracy instead of 7 for single precision. There are 2^{11} possible exponents in double precision, ranging from 1023 to -1022. The largest double precision number is of the order of $2^{1023} \approx 10^{307}$. Not only is double precision arithmetic more accurate than single precision, but the range of numbers is far greater.

Many features of IEEE arithmetic are illustrated in the program below. The reader would do well do do some of this kind of experimentation for herself or himself. The first section illustrates various how `inf` and `NaN` work. Note that e^{204} = `inf` in single precision but not in double precision because the range of values is larger in double precision. The next section shows that IEEE addition is commutative. This is a consequence of the "exact answer correctly rounded" procedure. The exact answer is commutative so either way the computer have the same number to round. This commutativity does not apply to triples of numbers because the computer only adds two numbers at a time. The compiler turns the expression `x + y + z` into `(x + y) + z`. It adds `x` to `y`, rounds the answer and adds the result to `z`. The expression `z + x + y` causes `z + x` to be rounded and added to `y`, which gives a (slightly) different result. Then comes an illustration of type conversion. Doing the integer division `i/j` gives 1/2 which is rounded to the integer value, 0, and then converted to floating point format. When `y` is computed, the conversion to floating point format is done before the division. Finally there is another example in which two variables might be expected to be equal but aren't because of inexact arithmetic. It is almost always wrong to ask whether two floating point numbers are equal. Last is a serendipitous example of getting exactly the right answer by accident. Don't count on this!

```
c   A program that explores IEEE arithmetic

c       Some double precision (8 byte or 64 bit) variables

      double precision xd, yd

c                 C programmers note that other variables are declared
c                 automatically and have default types.


c             Take an exponential that is out of range

      x = 204.
      y = exp(x)
      write(6,20) x,y
   20 format(´ The exponential of ´,e12.4,´   is ´, e12.4)

c             Divide a normal number by infinity

      z = x/y
      write(6,40) x, y, z
   40 format(´ ´,e12.4,´ divided by ´,e12.4,´ gives ´,e12.4)

c             Divide infinity by infinity

      w = y
      z = w/y
      write(6,60) w, y, z
   60 format(´ ´,e12.4,´ divided by ´,e12.4,´ gives ´,e12.4)

c             Take the square root of a negative number

      z = sqrt( -1. )
      write(6,80) z
   80 format(´ The square root of -1. is ´,e12.4)

c              Add NaN to a normal number

      w = x + z
      write(6,100) x, z, w
  100 format(´ ´,e12.4,´ plus ´, e12.4,´ gives ´,e12.4)

c              Take the same exponential in double precision
```

```
      xd = 204.d0
      yd = exp( xd )
      write(6,120) xd, yd
  120 format(´ The double precision exponential of ´,e12.4,´ is ´,e12.4)

c              Truncate a very large double precision number

      y = yd
      write(6,140) yd, y
  140 format(´ The single precision version of ´,e12.4,´ is ´,e12.4)

c             Explore the arithmetic, does addition commute?

      x =      sin(1.)
      y =  10.*sin(2.)
      z = 100.*sin(3.)
      s1 = x + y
      s2 = y + x
      s3 = x + z
      s4 = z + x
      if ( ( s1 .eq. s2 ) .and. ( s3 .eq. s4 ) ) then
         write (6,160) x, y, z
  160    format(´ Adding pairs of   ´,e12.4,´, ´,e12.4,´, and ´,e12.4,
     .                                       ´ commutes´)
      else
         write (6,180) x, y, z
  180    format(´ Adding pairs of ´,e12.4,´, ´,e12.4,´, and ´,e12.4,
     .                                       ´ does not commute´)
      endif

      s1 = x + y + z
      s2 = y + z + x
      s3 = z + x + y
      if ( ( s1 .eq. s2 ) .and. ( s2 .eq. s3 ) ) then
         write (6,200) x, y, z
  200    format(´ Adding triples of ´,e12.4,´, ´,e12.4,´, and ´,e12.4,
     .                                       ´ commutes´)
      else
         write (6,220) x, y, z
  220    format(´ Adding triples of ´,e12.4,´, ´,e12.4,´, and ´,e12.4,
     .                                       ´ does not commute´)
      endif
```

```
c             The result of an operation depends on the type of the operand

      i = 1
      j = 2
      x = i/j
      y = real(i)/real(j)
      write(6,240) i,j,x,y
  240 format(´ Dividing ´,i4,´ by ´,i4,´ gives ´,e12.4,
     .                ´ without conversion´,/,
     .  ´                          and ´,e12.4,
     .                ´ with conversion´)

c         Don´t expect other arithmetic identities to be exactly true.

      x = z / 11
      y = x + x + x + x + x + x + x + x + x + x + x
      x = z / 5.
      w = 5*x
      if ( y .eq. z ) then
         write(6,260) z
  260    format(´ Divided ´,e12.5,´ by 11 and got it back ´,
     .                                 ´by addition!!!´)
      else
         write(6,280) z
  280    format(´ Divided ´,e12.5,´ by 11 and did not get it back ´,
     .                                 ´by addition´)
      endif
      if ( w .eq. z ) then
         write(6,300) z
  300    format(´ Divided ´,e12.5,´ by  5 and got it back ´,
     .                                 ´by multiplication!!!´)
      else
         write(6,320) z
  320    format(´ Divided ´,e12.5,´ by  5 and did not get it back ´,
     .                                 ´by multiplication´)
      endif

      stop
      end
```

This program produces the output

```
 The exponential of   0.2040E+03   is Inf
   0.2040E+03 divided by Inf          gives   0.0000E+00
```

```
Inf          divided by Inf          gives NaN
The square root of -1. is NaN
  0.2040E+03 plus NaN          gives NaN
The double precision exponential of   0.2040E+03 is   0.3945E+89
The single precision version of   0.3945E+89 is Inf
Adding pairs of     0.8415E+00,   0.9093E+01, and   0.1411E+02 commutes
Adding triples of   0.8415E+00,   0.9093E+01, and   0.1411E+02 does not commute
Dividing    1 by    2 gives   0.0000E+00 without conversion
                        and   0.5000E+00 with conversion
Divided  0.14112E+02 by 11 and did not get it back by addition
Divided  0.14112E+02 by  5 and got it back by multiplication!!!
Note: the following IEEE floating-point arithmetic exceptions
occurred and were never cleared; see ieee_flags(3M):
Inexact;  Overflow;  Invalid Operand;
Note: IEEE Infinities were written to ASCII strings or output files; see econvert(3).
Note: IEEE NaNs were written to ASCII strings or output files; see econvert(3).
Sun´s implementation of IEEE arithmetic is discussed in
the Numerical Computation Guide.
```