The Perils of Floating Point

by Bruce M. Bush

Many great engineering and scientific advances of recent decades would not have been possible without the floating-point capabilities of digital computers. Still, some results of floating-point calculations look pretty strange, even to people with years of mathematical experience. I will attempt to explain the causes of some of these strange results and give some suggestions where appropriate.

Floating-point representations and arithmetic are inexact, but I don't believe that is particularly troublesome to most programmers. Many input values are measurements, which are inherently inexact, so the question about the output values isn't whether there is error, but how much error should be expected. However, when you can compute a more accurate result in your head than your computer can with its floating-point, you start to get suspicious.

I have programmed my examples in FORTRAN for a couple of reasons:
1. More floating-point calculations are performed in FORTRAN than any other computer language.
2. I work for a company, Lahey Computer Systems, that develops and sells FORTRAN language systems.

Binary Floating-Point

At the heart of many strange results is one fundamental: floating-point on computers is usually base 2, whereas the external representation is base 10. We expect that 1/3 will not be exactly representable, but it seems intuitive that .01 would be. Not so! .01 in IEEE single-precision format is exactly 10737418/1073741824 or approximately 0.009999999776482582. You might not even notice this difference until you see a bit of code like the following:

```
REAL X
DATA X /.01/
IF ( X * 100.d0 .NE. 1.0 ) THEN
   PRINT *, 'Many systems print this surprising result. '
ELSE
   PRINT *, 'And some may print this.'
ENDIF
```

Base-10 floating-point implementations don't have this anomaly. However, base-10 floating-point implementations are rare because base-2 (binary) arithmetic is so much faster on digital computers.

Inexactness

Floating-point arithmetic on digital computers is inherently inexact. The 24 bits (including the hidden bit) of mantissa in a 32-bit floating-point number represent approximately 7 significant decimal digits. Unlike the real number system, which is continuous, a floating-point system has gaps between each number. If a number is not exactly representable, then it must be approximated by one of the nearest representable values.

Because the same number of bits are used to represent all normalized numbers, the smaller the exponent, the greater the density of representable numbers. For example, there are approximately 8,388,607 single-precision numbers between 1.0 and 2.0, while there are only about 8191 between 1023.0 and 1024.0.

On any computer, mathematically equivalent expressions can produce different values using floating-point arithmetic. In the following example, Z and Z1 will typically have different values because (1/Y) or 1/7 is not exactly representable in binary floating-point:

```
REAL X, Y, Y1, Z, Z1
DATA X/77777/, Y/7/
Y1 = 1 / Y
Z = X / Y
Z1 = X * Y1
IF (Z .NE. Z1) PRINT *, 'Not equal!'
END
```

Insignificant Digits

The following code example illustrates the phenomenon of meaningless digits that could seem to be significant:

```
REAL A, Y
DATA Y /1000.2/     ! About 7 digits of precision in Y
A = Y - 1000.0      ! About 3 significant digits in result
PRINT *, A          ! Prints 0.200012
END
```

A single-precision (REAL) entity can represent a maximum of about 7 decimal digits of precision, so the subtraction above represents (1000.200 - 1000.000). The result, therefore, can only represent about 3 decimal digits. The program, however, will happily print out "0.200012". Because 1000.2 is not exactly representable in binary floating-point and 1000.0 is, the result A is a little larger than 0.2. The computer doesn't know that the digits beyond ".200" have no meaning.

Perhaps someday the computer will keep track of the number of bits in a result that are truly significant. For now, it is still the responsibility of the programmer. If you stay aware of the number of decimal digits represented by a data type, approximating the number of significant digits is a straight-forward, but perhaps time-consuming, task. Give the most attention to:

1. subtractions of numbers that are nearly equal,
2. additions of numbers whose magnitudes are nearly equal, but whose signs are opposite, and
3. additions and subtractions of numbers that differ greatly in magnitude.

Crazy Conversions

Conversions to integer can unmask inaccuracies in a floating-point number, as is demonstrated by the next example. The closest single-precision floating-point number to 21.33 is slightly less than 21.33, so when it is multiplied by 100., the result Y is slightly less than 2133.0. If you print Y in a typical floating-point format, rounding causes it to be displayed as 2133.00. However, if you

assign Y to an integer I, no rounding is done, and the number is truncated to 2132.

```
REAL X, Y
INTEGER I
X = 21.33       ! Slightly less than 21.33
Y = X * 100.    ! Slightly less than 2133.0
I = Y           ! Truncates to 2132
PRINT *, Y, I   ! Prints "2133.00      2132"
END
```

The following program prints "1.66661000251770" when compiled with Lahey's LF90:

```
DOUBLE PRECISION D
REAL X
X = 1.66661     ! Assign to single precision
D = X           ! Convert to double precision
PRINT *, D
END
```

You ask, "Why do you extend the single-precision number with the seemingly random '000251770'?" Well, the number isn't extended with random values; the computer's floating-point does the conversion by padding with zeros in the binary representation. So D is exactly equal to X, but when it is printed out to 15 decimal digits, the inexactness shows up. This is also another example of insignificant digits. Remember that assigning a single-precision number to a double-precision number doesn't increase the number of significant digits.

Too Many Digits

You may decide to check the previous program example by printing out both D and X in the same format, like this:

```
30  FORMAT (X, 2G25.15)
PRINT 30, X, D
```

In some FORTRAN implementations, both numbers print out the same. You may walk away satisfied, but you are actually being misled by low-order digits in the display of the single-precision number. In Lahey FORTRAN the numbers are printed out as:

```
     1.66661000000000         1.66661000251770
```

The reason for this is fairly simple: the formatted-I/O conversion routines know that the absolute maximum decimal digits that have any significance when printing a single-precision entity is 9. The rest of the field is filled with the current "precision-fill" character, which is "0" by default. The precision-fill character can be changed to any ASCII character, e.g., asterisk or blank. Changing the precision-fill character to "*" emphasizes the insignificance of the low-order digits:

```
     1.66661000******         1.66661000251770
```

Too Much Precision

The IEEE single-precision format has 24 bits of mantissa, 8 bits of exponent, and a sign bit. The internal floating-point registers in Intel microprocessors such as the Pentium have 64 bits of mantissa, 15 bits of exponent and a sign bit. This allows intermediate calculations to be performed with much less loss of precision than many other implementations. The down side of this is that, depending upon how intermediate values are kept in registers, calculations that look the same can give different results.

In the following example, a compiler could generate code that calculates A/B, stores the intermediate result into a single-precision temporary, calculates X/Y, performs a reversed subtract of the temporary, then stores the result. Z will not be zero, because precision will be lost in storing into a single-precision temporary. If the generated code keeps the intermediate result in registers, no precision will be lost, and Z will be zero.

```
REAL A, B, X, Y, Z
DATA A/10./, B/3./, X/10./, Y/3./
Z = (A/B) - (X/Y)
PRINT *, Z          ! Could be zero or not.
END
```

The next example illustrates a variation on the previous example. A compiler can still generate code that keeps the intermediate result C in a register, which means that Z will be zero. If precision is lost by storing A/B into C, then Z will be nonzero.

```
REAL A, B, C, X, Y, Z
DATA A/10./, B/3./, X/10./, Y/3./
C = A/B
Z = C - (X/Y)
PRINT *, Z          ! Could be zero or not.
END
```

The slight variation of adding the statement label 100 foils the optimization of keeping C in a register, so Z will probably be nonzero with almost any compiler.

```
REAL A, B, C, X, Y, Z
DATA A/10./, B/3./, X/10./, Y/3./
C = A/B
100  Z = C - (X/Y)
PRINT *, Z
IF ( ... ) GO TO 100
END
```

Safe Comparisons

Different computers use different numbers of bits to store floating-point numbers. Even when the same IEEE formats are used for storing numbers, differences in calculations can occur because of the size of intermediate registers. To increase portability and to ensure consistent results, I recommend against comparing for exact equality of real numbers in FORTRAN. A better technique is to compare the absolute value of the difference of two numbers with an appropriate epsilon to get relations like approximately equal, definitely greater than, etc.
Example:

```
REAL EPSILON, X, A, B
PARAMETER (EPSILON = .000001)
DATA A/13.9/, B/.000005/
X = (A * B) / B
IF (ABS(X - A) .LE. (ABS(X)*EPSILON)) THEN
   PRINT *, 'X is approximately equal to A'
```

```
ENDIF
IF ((X - A) .GT. (ABS(X)*EPSILON)) THEN
   PRINT *, 'X is definitely greater than A'
ENDIF
IF ((A - X) .GT. (ABS(X)*EPSILON)) THEN
   PRINT *, 'X is definitely less than A'
ENDIF
END
```

Multiplying the epsilon by one of the comparands adjusts the comparison to the range of the numbers, allowing a single epsilon to be used for many, or perhaps all compares. For the most predictable results, use an epsilon half as large and multiply it by the sum of the comparands, as in the following example:

```
REAL EPSILON, X, A, B
PARAMETER (EPSILON = .0000005) ! Smaller epsilon
DATA A/13.9/, B/.000005/
X = (A * B) / B
IF (ABS(X - A) .LE. (ABS(X+A)*EPSILON)) THEN
   PRINT *, 'X is approximately equal to A'
ENDIF
```

Even comparisons of greater-than, less-than-or-equal-to, etc., can produce unexpected results, because a floating-point computation can produce a value that is not mathematically possible. In the following example X is always mathematically greater than J, so X/J should always be greater than 1.0. For large values of J, however, the addition of delta is not representable by X, because of limited mantissa size.

```
REAL X, DELTA
DATA DELTA/.001/
DO 10 J = 1, 100 000
  X = J + DELTA               ! Make X bigger than J
  CALL SUB (X)                ! Force X out of register
  IF ( X/J .LE. 1.0 ) THEN      ! X/J always > 1 ?
   PRINT *, 'Error !'
    STOP
  END IF
10  CONTINUE
END
SUBROUTINE SUB (X)
END
```

Programming with the Perils

There are no easy answers. It is the nature of binary floating-point to behave the way I have described. In order to take advantage of the power of computer floating-point, you need to know its limitations and work within them. Keeping the following things in mind when programming floating-point arithmetic should help a lot:

1. Only about 7 decimal digits are representable in single-precision IEEE format, and about 16 in double-precision IEEE format.
2. Every time numbers are transferred from external decimal to internal binary or vice-versa, precision can be lost.
3. Always use safe comparisons.
4. Beware of additions and subtractions that can quickly erode the true significance in a result. The computer doesn't know what bits are truely significant. 5. Conversions between data types can be tricky. Conversions to double-precision don't increase the number of truely significant bits. Conversions to integer always truncate toward zero, even if the floating-point number is printed as a larger integer.
6. Don't expect identical results from two different floating-point implementations.

I hope that I have given you a little more awareness of what is happening in the internals of floating-point arithmetic, and that some of the strange results you have seen make a little more sense.

While some of the "perils" can be avoided, many just need to be understood and accepted.

IEEE Standard Floating-Point Formats

The Institute of Electrical and Electronics Engineers, Inc. (IEEE) has defined standards for floating-point representations and computational results (IEEE Std 754-1985). This section is an overview of the IEEE standard for representing floating-point numbers. The data contained herein helps explain some of the details in the rest of the article, but is not required for understanding the basic concepts.

Most binary floating-point numbers can be represented as 1.ffffff x 2^n, where the 1 is the integer bit, the f's are the fractional bits, and the n is the exponent. The combination of the integer bit and the fractional bits is called the mantissa (or significand). Because most numbers can have their exponent adjusted so that there is a 1 in the integer bit (a process called normalizing), the 1 does not need to be stored, effectively allowing for an extra bit of precision. This bit is called a hidden bit. Numbers are represented as sign-magnitude, so that a negative number has the same mantissa as a positive number of the same magnitude, but with a sign bit of 1. A constant, called a bias, is added to the exponent so that all exponents are positive.

The value 0.0, represented by a zero exponent and zero mantissa, can have a negative sign. Negative zeros have some subtle properties that will not be evident in most programs. A zero exponent with a nonzero mantissa is a "denormal." A denormal is a number whose magnitude is too small to be represented with an integer bit of 1 and can have as few as one significant bit.

Exponent fields of all ones (largest exponent) represent special numeric results. A mantissa of zero represents infinity (positive or negative); a nonzero mantissa represents a NAN (not-a-number). NANs, which occur as a result of invalid numeric operations, are not discussed further in this article.

The IEEE Standard defines 32-bit and 64-bit floating-point representations. The 32-bit (single-precision) format is, from high-order to low-order, a sign bit, an 8-bit exponent with a bias of 127, and 23 bits of mantissa. The 64-bit (double-precision) format is, a sign bit, an 11-bit exponent with a bias of 1023, and 52 bits of mantissa. With the hidden bit, normalized numbers have an effective precision of 24 and 53 bits, respectively.

Single-precision format
31, 30-23, 22-0
S, Exponent, Significand

Double-precision format
63, 62-52, 51-0
S, Exponent, Significand

Bibliography

American National Standards Institute (1978), "American National Standard, Programming Language FORTRAN", ANSI

X3.9-1978, ISO 1539-1980 (E).

IEEE Computer Society (1985), "IEEE Standard for Binary Floating-Point Arithmetic", IEEE Std 754-1985.