Floating point

From Wikipedia, the free encyclopedia

Floating-point is a numeral-interpretation system in which a string of digits (or bits) represents a real number. A system of arithmetic is defined that allows these representations to be manipulated with results that are similar to the arithmetic operations over real numbers. The representation uses an explicit designation of where the radix point (decimal point, or, more commonly in computers, binary point) is to be placed relative to that string. The designated location of the radix point is permitted to be far to the left or right of the digit string, allowing for the representation of very small or very large numbers. Floating-point could be thought of as a computer realization of scientific notation.

The large dynamic range of floating-point numbers frees the programmer from explicitly encoding number representations for their particular application. For this reason computation with floating-point numbers plays a very important role in an enormous variety of applications in science, engineering, and industry, particularly in meteorology, simulation, and mechanical design. The ability to perform floating point operations is an important measure of performance for computers intended for such applications. It is measured in "MegaFLOPS" (million FLoating-point Operations Per Second), or Gigaflops, etc. World-class supercomputer installations are generally rated in Teraflops.

Contents

Overview

There are several mechanisms by which strings of digits can represent numbers:

- The most common way of interpreting the value of a string of digits, so trivial that people rarely think about it, is as an integer—the radix point is implicitly at the right end of the string.
- In common mathematical notation, the digit string can be of any length, and the location of the radix point is indicated by placing an explicit "point" character (dot or comma) there.
- In "Fixed-point" systems, some specific convention is made about where the radix point is located in the string. For example, the convention could be made that the string consists of 8 digits, with the point in the middle, so that "00012345" has a value of 1.2345.
- In scientific notation, a radix character is used, but it is placed wherever it is convenient, and separate information is provided telling how that placement differs from its "mathematically correct" location. This discrepancy is given in the form of multiplication by some power of the radix. For example, the revolution period of Jupiter's moon Io is 152853.5047s. This could be represented in scientific notation in many equivalent ways:
  - $1528.535047 \times 10^{2}$
  - $1.528535047 \times 10^{5}$
  - $0.00001528535047 \times 10^{10}$

- Floating-point notation generally refers to a system similar to scientific notation, but without use of a radix character. The location of the radix point is specified solely by separate "exponent" information. It can be thought of as being equivalent to scientific notation with the requirement that the radix point be effectively in a "standard" place. That place is often chosen as just after the leftmost digit. This article will follow that convention. Under that convention, the orbital period of Io is 1528535047 with an exponent of 5. That is, the "standard" place for the radix point is just after the first digit: 1.528535047, and the exponent designation indicates the radix point is actually 5 digits to the right of that, that is, the number is 10^5 times bigger than that.

By allowing the radix point to be anywhere, floating-point notation allows calculations over a wide range of magnitudes, within a fixed number of digits, while maintaining good accuracy. For example, in a decimal floating-point system with 3 digits, the multiplication that humans would write as

$$.12 \times .12 = .0144$$

would be expressed as

$$(1.20 \times 10^{-1}) \times (1.20 \times 10^{-1}) = (1.44 \times 10^{-2})$$

In a fixed-point system with the decimal point at the left, it would be

$$.120 \times .120 = .014$$

A digit of the result was lost because of the inability of the digits and decimal point to 'float' relative to each other within the digit string.

Nomenclature

In floating-point representation, the string of digits is called the significand, or sometimes the mantissa. The representation of the significand is defined by a choice of *base* or *radix*, and the number of digits stored in that base. Throughout this article the base will be denoted by *b*, and the number of digits (or the precision) by *p*. Historically, different bases have been used for floating-point, but almost all modern computer hardware uses base 2, or binary. Some examples in this article will be in base 10, the familiar decimal notation.

The representation also includes a number called the exponent. This records the position, or offset, of the window of digits into the number. This can also be referred to as the characteristic, or scale. The window always stores the most significant digits in the number, the first non-zero digits in decimal or bits in binary. The exponent is the power of the base by which the significand is multiplied.

Floating-point notation *per se* is generally used only in computers, because it holds no advantage over scientific notation for human reading. There are many ways to represent floating-point numbers in computers—floating-point is a generic term to describe number representations in computing that are used to implement the above system of arithmetic. A number representation (called a numeral system in maths) specifies some way of storing a number as a string of bits. The arithmetic is defined as a set of actions on bit-strings that simulate normal arithmetic operations. When the numbers being represented are the rationals, one immediate issue is that there are an infinite number of rational numbers, and only a finite number of bits inside a real computer. The numbers that we represent must be an approximation of the entire set. When we restrict ourselves to binary expansions of numbers (because these are easiest to operate upon in a digital computer) the subset of the Rationals that we operate on is restricted to denominators that are powers of 2. Now any rational with a denominator that has a factor other than 2 will have an infinite binary expansion. If we consider this expansion as an infinite string of bits then there are several methods for approximating this string in memory:

- When we store a fixed size window at a constant position in the bit-string the representation is called Fixed Point. The hardware to manipulate these representations is less costly than Floating-Point and is commonly used to perform integer operations. In this case the radix point is always beneath the window.
- When we store a fixed size window that is allowed to slide up and down the bit-string, with the radix point location not necessarily under the window, the representation is called **Floating-point**.

However, the most common ways of representing floating-point numbers in computers are the formats standardized as the IEEE 754 standard, commonly called "IEEE floating-point". These formats can be manipulated efficiently by nearly all modern floating-point computer hardware, and this article will focus on them. The standard actually provides for many closely-related formats, differing in only a few details. Two of these formats are ubiquitous in computer hardware and languages:

- Single precision, called "float" in the C language family, and "real" or "real*4" in Fortran. It occupies 32 bits (4 bytes) and has a significand precision of 24 bits. This gives it an accuracy of about 7 decimal digits.
- Double precision, called "double" in the C language family, and "doubleprecision" or "real*8" in Fortran. It occupies 64 bits (8 bytes) and has a significand precision of 53 bits. This gives it an accuracy of about 16 decimal digits.

Alternative computer representations for non-integral numbers

While the standard IEEE formats are by far the most common because they are efficiently handled in most computer processors, there are a few alternate representations that are sometimes used:

- Fixed-point representation uses integer hardware operations with a specific convention about the location of the binary point, for example, 6 bits from the left. This has to be done in the context of a program that makes whatever convention is required. It is usually used in special-purpose applications on embedded processors that can only do integer arithmetic.
- Where extreme precision is desired, floating-point arithmetic can be emulated in software with extremely large significand fields. The significands might grow and shrink as the program runs. This is called Arbitrary-precision arithmetic or "bignum" arithmetic.
- Some numbers (e.g. 1/3) can't be represented exactly in binary floating-point no matter what the precision. Software packages that perform rational arithmetic represent numbers as fractions with integral numerator and denominator, and can therefore represent any rational number exactly. Such packages generally need to use bignum arithmetic for the individual integers.
- Some software packages (e.g. Maxima and Maple) can perform symbolic arithmetic, handling irrational numbers like π or $\sqrt{3}$ in a completely "formal" way, without dealing with the bits of the significand. Such programs can evaluate "$\sqrt{3}$" or "sin3π" exactly, because they "know" the underlying mathematics.

Normalization

The requirement that the leftmost digit of the significand be nonzero, is called **normalization**. By doing this, one no longer needs to express the point explicitly; the exponent provides that information. In decimal floating-point notation with precision of 10, the revolution period of Io is simply an exponent e=5 and a significand s=1528535047. The implied decimal point is after the first digit of s (after the '1' and before the first '5').

When a (nonzero) floating-point number is normalized, its leftmost digit is nonzero. The value of the significand obeys $1 \leq s < b$. (Zero needs special treatment; this will be described below.)

Value

The mathematical value of a floating-point number, using the convention given above, is s.ssssssss...sss × b^e.

Equivalently, it is:

$\frac{s}{b^{p-1}} \times b^e$ (where **s** means the integer value of the entire significand)

In binary radix, the significand is a string of bits (1's and 0's) of length *p*, of which the leftmost bit is 1. The real number π, represented in binary as an infinite series of bits is

11.0010010000111111011010101000100010000101101000110000100011010011... but is
11.00100100001111110110101 when approximated by rounding to a precision of 24 bits.

In binary floating-point, this is e=1 ; s=11001001000011111011011. It has a decimal value of

3.141597410125732421875 (exactly!), whereas the true value of π is
3.14159265358979323846264338327 95...

The problem is that floating-point numbers with a limited number of digits can represent only a subset of the real numbers, so any real number outside of that subset (e.g. 1/3, or an irrational number such as π), cannot be represented exactly. Even numbers with extremely short decimal representations can suffer from this problem. The decimal number 0.1 is not representable in binary floating-point of any finite precision. The exact binary representation would have a "1100" sequence continuing endlessly:

e=-4; s=11001100110011001100110011001100110011..., but when rounded to 24 bits it becomes

e=-4; s=11001100110011001100110 which is actually 0.10000000149011611938476562 5 in decimal.

Conversion and rounding

When a number is represented in some other format (such as a string of digits), then it will require a conversion to be used in floating-point format. If the number can be represented in the floating point format then the conversion is exact. If there is not an exact representation then the conversion requires a choice of which float-point number is appropriate. There are several different rounding schemes for this decision that have been used. Originally truncation was the typical approach. Since the introduction of IEEE 754, the default method rounds to even, sometimes called Bankers Rounding. This method choses the nearest value, or in the case of a tie, so as to make the significand even. The result of rounding π to 24-bit binary floating-point differs from the true value by about 0.03 parts per million, and matches the decimal representation of π in the first 7 digits. The difference is the discretization error and is limited by the machine epsilon.

The other time that a rounding mode is used, is when the result of an operation on floating-point numbers has more significant digits than there are places in the significand. In this case the rounding mode is applied to the intermediate value as if it were being converted into a floating-point number. Other common rounding modes always round the number in a certain direction (e.g. towards zero). These alternative modes are useful when the amount of error being introduced must be known. Applications that require a known error are multi-precision floating-point, and interval arithmetic.

Mantissa

The word *mantissa* is often used as a synonym for significand. Purists may not consider this usage to be correct, since the mantissa is traditionally defined as the fractional part of a logarithm, while the *characteristic* is the integer part. This terminology comes from the way logarithm tables were used before computers became commonplace. Log tables were actually tables of mantissas. Therefore, a mantissa is the logarithm of the significand.

History

The floating point system of numbers was used by the Kerala School of mathematics in 14th century India to investigate and rationalise about the convergence of series.

In 1938, Konrad Zuse of Berlin, completed the "Z1", the first mechanical binary programmable computer. It was based on Boolean Algebra and had most of the basic ingredients of modern machines, using the binary system and today's standard separation of storage and control. Zuse's 1936 patent application (Z23139/GMD Nr. 005/021) also suggests a 'von Neumann' architecture (re-invented in 1945) with program and data modifiable in storage. Originally the machine was called the "V1" but retroactively renamed after the war, to avoid confusion with the V1 missile. It worked with floating point numbers having a 7-bit exponent, 16-bit mantissa, and a sign bit. The memory used sliding metal parts to store 16 such numbers, and worked well; but the arithmetic unit was less successful, occasionally suffering from certain mechanical engineering problems. The program was read from punched discarded 35 mm movie film. Data values could be entered from a numeric keyboard, and outputs were displayed on electric lamps. The machine was not a general purpose computer because it lacked looping capabilities. The Z3 was completed in 1941 and was program controlled.

Once electronic digital computers became a reality, the need to process data in this way was quickly recognized. The first commercial computer to be able to do this in hardware appears to be the Z4 in 1950, followed by the IBM 704 in 1954. For some time after that, floating-point hardware was an optional feature, and computers that had it were said to be "scientific computers", or to have "scientific computing" capability. All modern general-purpose computers have this ability. The PDP-11/44 was an extension of the 11/34 that included the cache memory and floating point units as a standard feature.

The UNIVAC 1100/2200 series, introduced in 1962, supported two floating point formats: single precision - 36 bits: 1 sign bit, 8 bit characteristic, 27 bit mantissa, and double precision - 72 bits: 1 sign bit, 11 bit characteristic, 60 bit mantissa. The IBM 7094, introduced the same year, also supported single and double precision, with slightly different format.

Prior to the IEEE-754 standard, computers had many different formats (and, indeed, different word sizes), and there were also multiple formats in software emulations.

The IEEE-754 standard was created in the early 1980s, after word sizes of 32 bits (or 16 or 64) had been generally settled upon. Among its innovations are these:

- A precisely specified encoding of the bits, so that all compliant computers would interpret bit patterns the same way. This

made it possible to transfer floating-point numbers from one computer to another.
- A precisely specified behavior of the arithmetic operations. This meant that a given program, with given data, would always produce the same result on any compliant computer. This helped reduce the almost mystical reputation that floating-point computation had for seemingly nondeterministic behavior.
- The ability of exceptional conditions (overflow, divide by zero, etc.) to propagate through a computation in a benign manner and be handled by the software in a controlled way.

Floating point arithmetic operations

The usual rule for performing floating point arithmetic is that the exact mathematical value is calculated[1], and the result is then rounded to the nearest representable value in the specified precision. This is in fact the behavior mandated for IEEE-compliant computer hardware, under normal rounding behavior and in the absence of exceptional conditions.

For ease of presentation and understanding, decimal radix with 7 digit precision will be used in the examples. The fundamental principles are the same in any radix or precision.

Addition

A simple method to add floating point numbers is to first represent them with the same exponent. In the example below, the second number is shifted right by three digits. We proceed with the usual addition method:

```
  e=5;  s=1.234567     (123456.7)
+ e=2;  s=1.017654     (101.7654)

  e=5;  s=1.234567
+ e=5;  s=0.001017654  (after shifting)
--------------------
  e=5;  s=1.235584654  (true sum: 123558.4654)
```

This is the true result, the exact sum of the operands. It will be rounded to seven digits and then normalized if necessary. The final result is e=5; s=1.235585, or 123558.5.

Note that the low 3 digits of the second operand (654) are essentially lost. This is round-off error. In serious cases, the sum of two numbers different than 0 may be equal to one of them:

```
  e=5;  s=1.234567
+ e=-3; s=9.876543

  e=5;  s=1.234567
+ e=5;  s=0.00000009876543 (after shifting)
----------------------
  e=5;  s=1.23456709876543 (true sum)
  e=5;  s=1.234567         (after rounding/normalization)
```

Another problem of loss of significance occurs when two close numbers are subtracted. e=5; s=1.235585 and e=5; s=1.234567 are representations of the rationals 123558.5467 and 123456.7.

```
  e=5;  s=1.235585
- e=5;  s=1.235567
----------------
  e=5;  s=0.000018
  e=0;  s=1.800000 (after rounding/normalization)
```

The best representation of this difference is e=0; s=1.846700, which differs more than 2.59% from e=0; s=1.800000. This is called *cancellation* and illustrates the danger in assuming that all of the digits of a computed result are meaningful. It occurs when symmetric numbers whose absolutes are of close magnitude.

Dealing with the consequences of these errors are topics in numerical analysis.

Multiplication

To multiply, the significands are multiplied while the exponents are added, and the result is rounded and normalized.

```
  e=3;  s=4.734612
× e=5;  s=5.417242
-----------------------
  e=8;  s=25.648538980104 (true product)
  e=8;  s=25.64854        (after rounding)
  e=9;  s=2.564854        (after normalization)
```

Division is done similarly, but that is more complicated.

There are no cancellation or absorption problems with multiplication or division, though small errors may accumulate as operations are performed repeatedly. In practice, the way these operations are carried out in digital logic can be quite complex. (see Booth's multiplication algorithm and digital division)[2]

Computer representation

Floating-point numbers are typically packed into a computer datum as the sign bit, the exponent field, and the significand (mantissa), from left to right. For the common formats they are apportioned as follows:

```
        sign   exponent  (exponent bias)  significand   total
single   1        8         (127)             23         32
double   1       11         (1023)            52         64
```

Since the exponent can be positive or negative, it has a fixed "bias" added to it, and the sum is stored in the actual exponent field as an unsigned number. A value of zero, or all 1's, in that field is reserved for special treatment. Therefore the legal exponent range for normalized numbers is [-126, 127] for single precision or [-1022, 1023] for double.

When a number is normalized, its leftmost **significand** bit is known to be 1. In the IEEE single and double precision formats that bit is not actually stored in the computer datum. It is called the "hidden" or "implicit" bit. Because of this, single precision format actually has 24 bits of significand precision, while double precision format has 53.

For example, it was shown above that π, rounded to 24 bits of precision, has:

- sign = 0 ; e=1 ; s=110010010000111111011011 (including the hidden bit)

The sum of the exponent bias (127) and the exponent (1) is 128, so this is represented in single precision format as

- 0 10000000 10010010000111111011011 (excluding the hidden bit) = 40490FDB in hexadecimal

Dealing with exceptional cases

Floating-point computation in a computer can run into two kinds of problems:

- An operation can be mathematically illegal, such as division by zero, or calculating the square root of -1 or the inverse sine of 2.
- An operation can be legal in principle, but the result can be impossible to represent in the specified format, because the exponent is too large or too small to encode in the exponent field. Such an event is called an overflow (exponent too large) or underflow (exponent too small.)

Prior to the IEEE standard, such things usually caused the program to terminate, or caused some kind of trap that the programmer might be able to catch. How this worked was system-dependent, meaning that floating-point programs were not portable. Modern IEEE-compliant systems have a uniform way of handling these situations. An important part of the mechanism involves *error values* that result from a failing computation, and that can propagate silently through subsequent computation until they are detected at a point of the programmer's choosing.

The two error values are "infinity" (often denoted "INF"), and "NaN" ("not a number"), which covers all other errors.

> "Infinity" does not necessarily mean that the result is actually infinite. It simply means "too large to represent".

Both of these are encoded with the exponent field set to all 1's. (Recall that exponent fields of all 0's or all 1's are reserved for special meanings.) The significand field is set to something that can distinguish them—typically zero for INF and nonzero for NaN. The sign bit is meaningful for INF, that is, floating-point hardware distinguishes between $+\infty$ and $-\infty$.

When a nonzero number is divided by zero (the divisor must be *exactly* zero), a "zerodivide" event occurs, and the result is set to infinity of the appropriate sign. In other cases in which the result's exponent is too large to represent, an "overflow" event occurs, also producing infinity of the appropriate sign.

> Division of an extremely large number by an extremely small number can overflow and produce infinity. This is different from a zerodivide, though both produce a result of infinity, and the distinction is usually unimportant in practice.

Floating-point hardware is generally designed to handle operands of infinity in a reasonable way, such as

- (+INF) + (+7) = (+INF)
- (+INF) × (-2) = (-INF)
- But: (+INF) × 0 = NaN—there is no meaningful thing to do

When the result of an operation has an exponent too small to represent properly, an "underflow" event occurs. The hardware responds to this by changing to a format in which the significand is not normalized, and there is no "hidden" bit—that is, all significand bits are represented. The exponent field is set to the reserved value of zero. The significand is set to whatever it has to be in order to be consistent with the exponent. Such a number is said to be "denormalized" (a "denorm" for short), or, in more modern terminology, "subnormal". Denorms are perfectly legal operands to arithmetic operations.

If no significant bits are able to appear in the significand field, the number is zero. Note that, in this case, the exponent field and significand field are all zeros—floating-point zero is represented by all zeros.

Other errors, such as division of zero by zero, or taking the square root of -1, cause an "operand error" event, and produce a NaN result. NaN's propagate aggressively through arithmetic operations—any NaN operand to any operation causes an operand error and produces a NaN result.

There are five special "events" that may occur, though some of them are quite benign:

- An overflow occurs as described previously, producing an infinity.
- An underflow occurs as described previously, producing a denorm or zero.
- A zerodivide occurs as described previously, producing an infinity of the appropriate sign.
- An "operand error" occurs as described previously, producing a NaN.
- An "inexact" event occurs whenever the rounding of a result changed that result from the true mathematical value. This occurs almost all the time, and is usually ignored. It is looked at only in the most exacting applications.

Computer hardware is typically able to raise exceptions when these events occur. How this is done is system-dependent. Usually these exceptions are all *masked* (disabled), relying only on the propagation of error values. Sometimes overflow, zerodivide, and operand error are enabled.

Implementation in actual computers

The IEEE has standardized the computer representation for binary floating-point numbers in IEEE 754. This standard is followed by almost all modern machines. Notable exceptions include IBM Mainframes, which support IBM's own format (in addition to IEEE 754 data types), and Cray vector machines, where the T90 series had an IEEE version, but the SV1 still uses Cray floating-point format.

The standard allows for many different precision levels, of which the 32 bit ("single") and 64 bit ("double") are by far the most common, since they are supported in common programming languages. Computer hardware (for example, the Intel Pentium series and the Motorola 68000 series) often provides an 80 bit extended precision format, with 15 exponent bits and 64 significand bits, with no hidden bit. There is controversy about the failure of most programming languages to make these extended precision formats available to programmers (although C and related programming languages usually provide these formats via the long double type on such hardware). System vendors may also provide additional extended formats (e.g. 128 bits) emulated in software.

As of 2000, the IEEE 754 standard is currently under revision. See IEEE 754r.

Behavior of computer arithmetic

The standard behavior of computer hardware is to round the ideal (infinitely precise) result of an arithmetic operation to the nearest representable value, and give that representation as the result. In practice, there are other options. IEEE-754-compliant hardware allows one to set the **rounding mode** to any of the following:

- round to nearest (the default; by far the most common mode)
- round up (toward $+\infty$; negative results round toward zero)
- round down (toward $-\infty$; negative results round away from zero)
- round toward zero (sometimes called "chop" mode; it is similar to the common behavior of float-to-integer conversions, which convert −3.9 to −3)

In the default rounding mode the IEEE 754 standard mandates the round-to-nearest behavior described above for all fundamental algebraic operations, including square root. ("Library" functions such as cosine and log are not mandated.) This means that IEEE-compliant hardware's behavior is completely determined in all 32 or 64 bits.

The mandated behavior for dealing with overflow and underflow is that the appropriate result is computed, taking the rounding mode into consideration, as though the exponent range were infinitely large. If that resulting exponent can't be packed into its field correctly, the overflow/underflow action described above is taken.

The arithmetical distance between two consecutive representable floating point numbers is called an "ULP", for Unit in the Last Place. For example, the numbers represented by 45670123 and 45670124 hexadecimal is one ULP. An ULP is about 10^{-7} in single precision, and 10^{-16} in double precision. The mandated behavior of IEEE-compliant hardware is that the result be within one-half of an ULP.

Accuracy problems

The facts that floating-point numbers cannot faithfully mimic the real numbers, and that floating-point operations cannot faithfully mimic true arithmetic operations, lead to many surprising situations.

For example, the non-representability of 0.1 and 0.01 means that the result of attempting to square 0.1 is neither 0.01 nor the representable number closest to it. In 24-bit (single precision) representation, 0.1 (decimal) was given previously as e=-4; s=11001100110011001100110, which is

.100000001490116119384765625 exactly.

Squaring this number gives

.010000000298023226097399174250313080847263336181640625 exactly.

Squaring it with single-precision floating-point hardware (with rounding) gives

.010000000707805156707763671875 exactly.

But the representable number closest to 0.01 is

.009999999776482582092285156250 exactly.

Also, the non-representability of π (and π/2) means that an attempted computation of tan(π/2) will not yield a result of infinity, nor will it even overflow. It is simply not possible for standard floating-point hardware to attempt to compute tan(π/2), because π/2 cannot be represented exactly. This computation in C:

```
// Enough digits to be sure we get the correct approximation.
double pi = 3.14159265358979323846264338327795;
double z = tan(pi/2.0);
```

Will give a result of 16331239353195370.0. In single precision (using the tanf function), the result will be -22877332.0.

By the same token, an attempted computation of sin(π) will not yield zero. The result will be (approximately) $.1225 \times 10^{-15}$ in double precision, or $-.8742 \times 10^{-7}$ in single precision. [3]

In fact, while addition and multiplication are both commutative (a+b = b+a and a×b = b×a), they are not associative (a + b) + c = a + (b + c). Using 7-digit decimal arithmetic:

```
1234.567 + 45.67844 = 1280.245
                      1280.245 + 0.0004 = 1280.245
but
45.67844 + 0.0004 = 45.67884
                    45.67884 + 1234.567 = 1280.246
```

They are also not distributive (a + b)×c = a×c + b×c :

```
1234.567 × 3.333333 = 4115.223
1.234567 × 3.333333 = 4.115223
                      4115.223 + 4.115223 = 4119.338
but
1234.567 + 1.234567 = 1235.802
                      1235.802 × 3.333333 = 4119.340
```

In addition to loss of significance, inability to represent numbers such as π and 0.1 exactly, and other slight inaccuracies, the following phenomena may occur:

- Cancellation: subtraction of nearly equal operands may cause extreme loss of accuracy. This is perhaps the most common and serious accuracy problem.
- Conversions to integer are unforgiving: converting (63.0/9.0) to integer yields 7, but converting (0.63/0.09) may yield 6. This is because conversions generally truncate rather than rounding.
- Limited exponent range: results might overflow yielding infinity, or underflow yielding a denormal value or zero. If a denormal number results, precision will be lost.
- Testing for safe division is problematical: Checking that the divisor is not zero does not guarantee that a division will not overflow and yield infinity.
- Equality is problematical! Two computational sequences that are mathematically equal may well produce different floating-point values. Programmers often perform comparisons within some tolerance (often a decimal constant, itself not accurately represented), but that doesn't necessarily make the problem go away.

Minimizing the effect of accuracy problems

Because of the problems noted above, naive use of floating point arithmetic can lead to many problems. A good understanding of numerical analysis is essential to the creation of robust floating point software. The subject is actually quite complicated, and the reader is referred to the references at the bottom of this article.

In addition to careful design of programs, careful handling by the compiler is essential. Certain "optimizations" that compilers might make (for example, reordering operations) can work against the goals of well-behaved software. There is some controversy about the failings of compilers and language designs in this area. See the external references at the bottom of this article.

Floating point arithmetic is at its best when it is simply being used to measure real-world quantities over a wide range of scales (such as the orbital period of Io or the mass of the proton), and at its worst when it is expected to model the interactions of quantities expressed as decimal strings that are expected to be exact. An example of the latter case is financial calculations. For this reason, financial software tends not to use a binary floating-point number representation. See: http://www2.hursley.ibm.com/decimal/. The "decimal" data type of the C# programming language, and the IEEE 854 standard, are designed to avoid the problems of binary floating point, and make the arithmetic always behave as expected when numbers are printed in decimal.

Double precision floating point arithmetic is more accurate than just about any physical measurement one could make. For example, it could indicate the distance from the Earth to the Moon with an accuracy of about 50 nanometers. So, if one were designing an integrated circuit chip with 100 nanometer features, that stretched from the Earth to the Moon, double precision arithmetic would be fairly close to being good enough.

What makes floating point arithmetic troublesome is that people write mathematical algorithms that perform operations an enormous number of times, and so small errors grow. A few examples are matrix inversion, eigenvector computation, and differential equation solving. These algorithms must be very carefully designed if they are to work well.

People often carry expectations from their mathematics training into the field of floating point computation. For example, it is known that $(x+y)(x-y) = x^2 - y^2$, and that $\sin^2\theta + \cos^2\theta = 1$, and that eigenvectors are degenerate if the eigenvalues are equal. These facts can't be counted on when the quantities involved are the result of floating point computation.

While a treatment of the techniques for writing high-quality floating-point software is far beyond the scope of this article, here are a few simple tricks:

The use of the equality test (`if (x==y) ...`) is usually not a good idea when it is based on expectations from pure mathematics. Such things are sometimes replaced with "fuzzy" tests (`if (abs(x-y) < 1.0E-13) ...`). The wisdom of doing this varies greatly. It is often better to organize the code in such a way that such tests are unnecessary.

An awareness of when loss of significance can occur is useful. For example, if one is adding a very large number of numbers, the individual addends are very small compared with the sum. This can lead to loss of significance. Suppose, for example, that one needs to add many numbers, all approximately equal to 3. After 1000 of them have been added, the running sum is about 3000. A typical addition would then be something like

```
3253.671
+  3.141276
--------
3256.812
```

The low 3 digits of the addends are effectively lost. The Kahan summation algorithm may be used to reduce the errors.

Another thing that can be done is to rearrange the computation in a way that is mathematically equivalent but less prone to error. As an example, Archimedes approximated π by calculating the perimeters of polygons inscribing and circumscribing a circle, starting with hexagons, and successively doubling the number of sides. The recurrence formula for the circumscribed polygon is:

$$t_0 = \frac{1}{\sqrt{3}}$$

$$t_{i+1} = \frac{\sqrt{t_i^2 + 1} - 1}{t_i} \qquad \text{second form :} \qquad t_{i+1} = \frac{t_i}{\sqrt{t_i^2 + 1} + 1}$$

$$\pi \sim 6 \times 2^i \times t_i, \qquad \text{converging as i} \to \infty$$

Here is a computation using IEEE "double" (53 bits of significand precision) arithmetic:

i	$6 \times 2^i \times t_i$, first form	$6 \times 2^i \times t_i$, second form
0	**3**.4641016151377543863	**3**.4641016151377543863
1	**3**.2153903091734710173	**3**.2153903091734723496
2	**3.1**59659420974940120	**3.1**59659420975006733
3	**3.14**60862151314012979	**3.14**60862151314352708
4	**3.142**71459964531363334	**3.142**71459964536892 25
5	**3.1418**730499801259536	**3.1418**730499798241950
6	**3.14166**27470548084133	**3.14166**27470568494473
7	**3.141610**1765997805905	**3.141610**1766046906629
8	**3.14159**70343230776862	**3.14159**70343215275928
9	**3.14159**37488171150615	**3.14159**37487713536668
10	**3.141592**9278733740748	**3.141592**9273850979885
11	**3.141592**7256228504127	**3.141592**7220386148377
12	**3.1415926**717412858693	**3.1415926**707019992125
13	**3.1415926**189011456060	**3.1415926**578678454728
14	**3.1415926**717412858693	**3.1415926**546593073709
15	**3.14159**19358822321783	**3.14159265**38571730119
16	**3.1415926**717412858693	**3.14159265**36566394222
17	**3.1415**810075796233302	**3.14159265**36065061913
18	**3.1415926**717412858693	**3.14159265**35939728836
19	**3.141**4061547378810956	**3.14159265**35908393901
20	**3.14**05434924008406305	**3.14159265**35900560168
21	**3.14**0006864691227361 7	**3.14159265**3589 8608396
22	**3.1**3494537565859299 19	**3.14159265**3589 8122118
23	**3.14**0006864691227361 7	**3.14159265**3589 7995552
24	**3**.2245152435345525443	**3.14159265**3589 768907
25		**3.14159265**3589 762246
26		**3.14159265**3589 762246
27		**3.14159265**3589 762246
28		**3.14159265**3589 762246
	The true value is	**3.14159265358979323 85**...

While the two forms of the recurrence formula are clearly equivalent, the first subtracts 1 from a number extremely close to 1, leading to huge cancellation errors. Note that, as the recurrence is applied repeatedly, the accuracy improves at first, but then it deteriorates. It never gets better than about 8 digits, even though 53-bit arithmetic should be capable of about 16 digits of precision. When the second form of the recurrence is used, the value converges to 15 digits of precision.

A few nice properties

One can sometimes take advantage of a few nice properties:

- Any integer less than or equal to 2^{24} can be exactly represented in the single precision format, and any integer less than or equal to 2^{53} can be exactly represented in the double precision format. Furthermore, any reasonable power of 2 times such a number can be represented. This property is sometimes used in purely integer applications, to get 53-bit integers on platforms that have double precision floats but only 32-bit integers.
- The bit representations are monotonic, as long as exceptional values are avoided and the signs are handled properly. Floating point numbers are equal if and only if their integer bit representations are equal. Comparisons for larger or smaller can be done with integer comparisons on the bit patterns, as long as the signs match. However, the actual floating point comparisons provided by hardware typically have much more sophistication in dealing with exceptional values.
- To a rough approximation, the bit representation of a floating point number is proportional to its base 2 logarithm, with an average error of about 3%. (This is because the exponent field is in the more significant part of the datum.) This can be exploited in some applications, such as volume ramping in digital sound processing.

See also

- Significant digits
- Fixed-point arithmetic
- Floating point value
- Computable number
- IEEE Floating Point Standard
- IBM Floating Point Architecture
- FLOPS
- −0 (number)
- half precision – single precision – double precision – quad precision – minifloat
- Scientific notation
- Numerical Recipes

Notes and references

1. ^ Computer hardware doesn't necessarily compute the exact value; it simply has to produce the equivalent rounded result as though it had computed the infinitely precise result.
2. ^ The enormous complexity of modern division algorithms once led to a famous error. An early version of the Intel Pentium chip was shipped with a division instruction that, on rare occasions, gave slightly incorrect results. Many computers had been shipped before the error was discovered. Until the defective computers were replaced, patched versions of compilers were developed that could avoid the failing cases. See "Pentium FDIV Bug"
3. ^ But an attempted computation of cos(π) yields -1 exactly. Since the derivative is nearly zero near π, the effect of the inaccuracy in the argument is far smaller than the spacing of the floating point numbers around -1, and the rounded result is exact.

External links

- An edited reprint of the paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys.
- David Bindel's Annotated Bibliography on computer support for scientific computation.
- Donald Knuth. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89684-2. Section 4.2: Floating Point Arithmetic, pp.214 – 264.
- Press et. al. *Numerical Recipes in C++. The Art of Scientific Computing,* ISBN 0-521-75033-4.
- Kahan, William and Darcy, Joseph (2001). How Java's floating-point hurts everyone everywhere. Retrieved Sep. 5, 2003 from http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.
- Introduction to Floating point calculations and IEEE 754 standard by Jamil Khatib
- Survey of Floating-Point Formats This page gives a very brief summary of floating-point formats that have been used over the years.

Retrieved from "http://en.wikipedia.org/wiki/Floating_point"

Categories: Pages needing expert attention from Computer science experts | Cleanup from August 2006 | Data types | Computer arithmetic