Jim Lambers
MAT 460/560
Fall Semester 2009-10
Lecture 5 Notes

These notes correspond to Section 1.2 in the text.

Roundoff Errors and Computer Arithmetic, cont'd

Machine Precision

In error analysis, it is necessary to estimate error incurred in each step of a computation. As such, it is desirable to know an upper bound for the relative error introduced by rounding. This leads to the following definition.

Definition *(Machine Precision) Let $\mathbb{F}$ be a floating-point number system. The* **unit roundoff** *or* **machine precision**, *denoted by ϵ_{mach}, is the real number that satisfies*

$$\left|\frac{\text{fl}(x) - x}{x}\right| \leq \epsilon_{\text{mach}}$$

for any real number x such that $\text{UFL} < x < \text{OFL}$.

An intuitive definition of ϵ_{mach} is that it is the smallest positive number such that

$$\text{fl}\left(1 + \epsilon_{\text{mach}}\right) > 1.$$

The value of ϵ_{mach} depends on the rounding strategy that is used. If rounding toward zero is used, then $\epsilon_{\text{mach}} = \beta^{1-p}$, whereas if rounding to nearest is used, $\epsilon_{\text{mach}} = \frac{1}{2}\beta^{1-p}$.

It is important to avoid confusing ϵ_{mach} with the underflow level UFL. The unit roundoff is determined by the number of digits in the mantissa, whereas the underflow level is determined by the range of allowed exponents. However, we do have the relation that $0 < \text{UFL} < \epsilon_{\text{mach}}$.

Example The following table summarizes the main aspects of a general floating-point system and a double-precision floating-point system that uses a 52-bit mantissa and 11-bit exponent. For both systems, we assume that rounding to nearest is used, and that normalization is used.

	General	Double Precision
Form of machine number	$\pm m\beta^E$	$\pm 1.d_1d_2\cdots d_{52}2^E$
Precision	p	53
Exponent range	$L \leq E \leq U$	$-1023 \leq E \leq 1024$
UFL (Underflow Level)	β^L	2^{-1023}
OFL (Overflow Level)	$\beta^{U+1}(1-\beta^{-p})$	$2^{1025}(1-2^{-53})$
ϵ_{mach}	$\frac{1}{2}\beta^{1-p}$	2^{-53}

The Matlab functions `eps`, `realmax` and `realmin` return the values of ϵ_{mach}, OFL and UFL, respectively, on the system on which Matlab is being used. On a machine using an Intel Pentium processor, Matlab returns the following values when these functions are called:

Function	Value (5-digit decimal, rounded)	Value (Exact)
`eps`	2.2204×10^{-16}	2^{-52}
`realmax`	1.7977×10^{308}	$2^{1024}(1 - 2^{-53})$
`realmin`	2.2251×10^{-308}	2^{-1022}

The UFL and OFL values differ from the ones in the previous table because the exponent of -1023 (corresponding to all zeros in the exponent) is used to represent zero, since zero cannot be represented when using normalization, and the exponent of 1024 (corresponding to all ones in the exponent) is used to represent ∞ and $-\infty$, so that computation does not have to halt after overflow occurs. The value returned by `eps` is not the true value of ϵ_{mach}, because it can be shown by experimentation that

$$\epsilon_{\text{mach}} \leq 2^{-53} + 2^{-105},$$

which is much closer to the value that is given in the first table. □

Floating-Point Arithmetic

We now discuss the various issues that arise when performing *floating-point arithmetic*, or *finite-precision arithmetic*, which approximates arithmetic operations on real numbers.

When adding or subtracting floating-point numbers, it is necessary to shift one of the operands so that both operands have the same exponent, before adding or subtracting the mantissas. As a result, digits of precision are lost in the operand that is smaller in magnitude, and the result of the operation cannot be represented using a machine number. In fact, if x is the smaller operand and y is the larger operand, and $|x| < |y|\epsilon_{\text{mach}}$, then the result of the operation will simply be y (or $-y$, if y is to be subtracted from x), since the entire value of x is lost in rounding the result.

In multiplication or division, the operands need not be shifted, but the mantissas, when multiplied or divided, cannot necessarily be represented using only p digits of precision. The product of two mantissas requires $2p$ digits to be represented exactly, while the quotient of two mantissas could conceivably require infinitely many digits. Furthermore, overflow or underflow may occur depending on the exponents of the operands, since their sum or difference may lie outside of the interval $[L, U]$.

Because floating-point arithmetic operations are not exact, they do not follow all of the laws of real arithmetic. In particular, floating-point arithmetic is not associative; i.e., $x+(y+z) \neq (x+y)+z$ in floating-point arithmetic.

Absolute Error and Relative Error

Now that we have been introduced to some specific errors that can occur during computation, we introduce useful terminology for discussing such errors. Suppose that a real number $\hat{y}$ is an approximation to some real number y. For instance, $\hat{y}$ may be the closest number to y that can be represented using finite precision, or $\hat{y}$ may be the result of a sequence of arithmetic operations performed using finite-precision arithmetic, where y is the result of the same operations performed using exact arithmetic.

Definition *(Absolute Error, Relative Error) Let $\hat{y}$ be a real number that is an approximation to the real number y. The* **absolute error** *in $\hat{y}$ is*

$$E_{abs} = \hat{y} - y.$$

The **relative error** *in $\hat{y}$ is*

$$E_{rel} = \frac{\hat{y} - y}{y},$$

provided that y is nonzero.

The absolute error is the most natural measure of the accuracy of an approximation, but it can be misleading. Even if the absolute error is small in magnitude, the approximation may still be grossly inaccurate if the exact value y is even smaller in magnitude. For this reason, it is preferable to measure accuracy in terms of the relative error.

The magnitude of the relative error in $\hat{y}$ can be interpreted as a percentage of $|y|$. For example, if the relative error is greater than 1 in magnitude, then $\hat{y}$ can be considered completely erroneous, since the error is larger in magnitude as the exact value. Another useful interpretation of the relative error concerns *significant digits*, which are all digits excluding leading zeros. Specifically, if the relative error is at most β^{-p}, where β is an integer greater than 1, then the representation of $\hat{y}$ in base β has at least p correct significant digits.

It should be noted that the absolute error and relative error are often defined using absolute value; that is,

$$E_{abs} = |\hat{y} - y|, \quad E_{rel} = \left| \frac{\hat{y} - y}{y} \right|.$$

This definition is preferable when one is only interested in the magnitude of the error, which is often the case. If the sign, or direction, of the error is also of interest, then the first definition must be used.

Example Assume that we are using a floating-point system that uses base $\beta = 10$ (decimal), with 4 digits of precision and rounding to nearest. Then, if we add the numbers 0.4567×10^0 and 0.8580×10^{-2}, we obtain the exact result

$$x = 0.4567 \times 10^0 + 0.008530 \times 10^0 = 0.46523 \times 10^0,$$

which is rounded to

$$\text{fl}(x) = 0.4652 \times 10^0.$$

The absolute error in this computation is

$$E_{abs} = \text{fl}(x) - x = 0.4652 - 0.46523 = -0.00003,$$

while the relative error is

$$E_{rel} = \frac{\text{fl}(x) - x}{x} = \frac{0.4652 - 0.46523}{0.46523} \approx -0.000064484.$$

Using the same floating-point system, suppose that we multiply 0.4567×10^4 and 0.8530×10^{-2}. The exact result is

$$x = (0.4567 \times 10^4) \times (0.8530 \times 10^{-2}) = 0.3895651 \times 10^2 = 38.95651,$$

which is rounded to

$$\text{fl}(x) = 0.3896 \times 10^2 = 38.96.$$

The absolute error in this computation is

$$E_{abs} = \text{fl}(x) - x = 38.96 - 38.95651 = 0.00349,$$

while the relative error is

$$E_{rel} = \frac{\text{fl}(x) - x}{x} = \frac{38.96 - 38.95651}{38.95651} \approx 0.000089587.$$

We see that in this case, the relative error is smaller than the absolute error, because the exact result is larger than 1, whereas in the previous operation, the relative error was larger in magnitude, because the exact result is smaller than 1.

It should be noted that given a number such as 0.4567×10^4, the actual mantissa is the number 4.567 and the exponent is 3, since the mantissa m in a normalized floating-point system must satisfy $1 \leq m < \beta$, where β is the base. In this example, the numbers are written in such a way that all four significant digits are grouped together on the same side of the decimal point. The examples in the text use this format for examples involving decimal floating-point arithmetic, so it is used here as well for consistency. □

Example Suppose that the exact value of a computation is supposed to be 10^{-16}, and an approximation of 2×10^{-16} is obtained. Then the absolute error in this approximation is

$$E_{abs} = 2 \times 10^{-16} - 10^{-16} = 10^{-16},$$

which suggests the computation is accurate because this error is small. However, the relative error is

$$E_{rel} = \frac{2 \times 10^{-16} - 10^{-16}}{10^{-16}} = 1,$$

which suggests that the computation is completely erroneous, because by this measure, the error is equal in magnitude to the exact value; that is, the error is 100%. It follows that an approximation of zero would be just as accurate. This example, although an extreme case, illustrates why the absolute error can be a misleading measure of error. □

Cancellation

Subtraction of floating-point numbers presents a unique difficulty, in addition to the rounding error previously discussed. If the operands, after shifting exponents as needed, have leading digits in common, then these digits cancel and the first digit in which the operands do not match becomes the leading digit. However, since each operand is represented using only p digits, it follows that the result contains only $p - m$ correct digits, where m is the number of leading digits that cancel.

In an extreme case, if the two operands differ by less than ϵ_{mach}, then the result contains no correct digits; it consists entirely of roundoff error from previous computations. This phenomenon is known as *catastrophic cancellation*. Because of the highly detrimental effect of this cancellation, it is important to ensure that no steps in a computation compute small values from relatively large operands. Often, computations can be rearranged to avoid this risky practice.

Example Consider the quadratic equation

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

which has the solutions

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Suppose that $b > 0$. Then, in computing x_1, we encounter catastrophic cancellation if b is much larger than a and c, because this implies that $\sqrt{b^2 - 4ac} \approx b$ and as a result we are subtracting two numbers that are nearly equal in computing the numerator. On the other hand, if $b < 0$, we encounter this same difficulty in computing x_2.

Suppose that we use 4-digit rounding arithmetic to compute the roots of the equation

$$x^2 + 10,000x + 1 = 0.$$

Then, we obtain $x_1 = 0$ and $x_2 = -10,000$. Clearly, x_1 is incorrect because if we substitute $x = 0$ into the equation then we obtain the contradiction $1 = 0$. In fact, if we use 7-digit rounding arithmetic then we obtain the same result. Only if we use at least 8 digits of precision do we obtain roots that are reasonably correct,

$$x_1 \approx -1 \times 10^{-4}, \quad x_2 \approx -9.9999999 \times 10^3.$$

A similar result is obtained if we use 4-digit rounding arithmetic but compute x_1 using the formula

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}},$$

which can be obtained by multiplying and dividing the original formula for x_1 by the *conjugate* of the numerator, $-b - \sqrt{b^2 - 4ac}$. The resulting formula is not susceptible to catastrophic cancellation, because an addition is performed instead of a subtraction. □

Other Arithmetic Systems

It is natural to ask whether it is feasible to use *variable-precision* arithmetic, in which the precision is automatically increased to represent results of operations exactly. Variable-precision arithmetic has been implemented in some software environments, such as Maple. Unfortunately, the increased accuracy is more than offset by the loss of efficiency. Not only are arithmetic operations more expensive due to the increased precision, but they are implemented in software, rather than hardware, because hardware only performs arithmetic in fixed precision. As a result, variable-precision arithmetic does not enjoy widespread use.

To aid in error estimation, some arithmetic systems perform *interval analysis* on arithmetic operations that are implemented using either fixed or variable precision. In interval analysis, the result of each arithmetic operation is represented not as a single floating-point number but as an interval $[a, b]$ such that the result that would be obtained using exact arithmetic lies within the interval $[a, b]$. This interval is used to determine the appropriate interval for future computations that use the associated result as an operand. In the case of variable-precision arithmetic, interval analysis can increase efficiency by providing a guide to precision adjustment: if the interval becomes too large, then precision should be increased, perhaps repeating computations in order to ensure sufficient accuracy.