2 Computation with Floating-Point Numbers

2.1 Floating-Point Representation

The notion of real numbers in mathematics is convenient for hand computations and formula manipulations. However, real numbers are not well-suited for computation on a calculator or computer, because their numerical representation as a string of digits expressed in, say, base 10 can be very long or even infinitely long. Examples include $\pi, \sqrt{2}, e$, and $1/3$. In practice, computers store numbers with finite precision. Numbers and arithmetic used in scientific computation should meet a few general criteria:

- Numbers should have modest storage requirements.
- Arithmetic operations should be efficient to carry out.
- A level of standardization, or portability, is desirable. Results obtained on one computer should closely match the results of the same computation on other computers.

Standardized methods for representing numbers on computers have been established by the Institute of Electrical and Electronics Engineers (IEEE) to satisfy these basic goals. This lecture is concerned with *floating-point numbers*. The floating-point we will consider are available on most modern computing systems.[1] They are sometimes referred to as *double precision floating-point numbers*. A precise definition is given below. We comment on other kinds of floating-point numbers used in computers at the end of this section. In this lecture the term "floating-point number" always means a number that can be exactly represented by a double-precision floating-point number.

Differently from real numbers, there are only finitely many floating-point numbers. In particular, there is a smallest (and largest) floating-point number. In between there are also necessarily many gaps of numbers that cannot be represented exactly.

In hand computations, we usually represent numbers in terms of decimal digits. For instance,

$$x = 1.29 \cdot 10^2 \tag{1}$$

is one way to write the number 129. This way of writing numbers is sometimes called *scientific notation*. The 1.29 part is called the *mantissa*, 10 the *base*, and 2 the *exponent*. Another way to write x is $1 \cdot 10^2 + 2 \cdot 10^1 + 9 \cdot 10^0$. If, like in equation (1), the mantissa always has 3 decimal digits and the exponent has one decimal digit, then the distance between the number 1, which can be represented exactly, and a closest different representable number, $0.99 \cdot 10^0$, is $1 \cdot 10^{-2}$. The largest representable number in this scheme is $9.99 \cdot 10^9$ and the closest distinct positive number is $9.98 \cdot 10^9$. Their distance is $9.99 \cdot 10^9 - 9.98 \cdot 10^9 = 1 \cdot 10^7$, which is large. Thus, the distance

[0] Version September 9, 2013

[1] Notable exceptions include graphics processors.

between distinct representable numbers grows with their magnitude; it can be large for numbers of large magnitude.

Similarly to (1), in base 2, the number

$$y = 1.01 \cdot 2^2$$

may be a suitable representation of $101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, which is a binary representation of the number 5 in base 10. A binary digit is referred to as a *bit*.

Computers represent floating-point numbers using base 2. Floating-point numbers are generally stored in the form

$$x = \pm(1 + f) \cdot 2^e, \tag{2}$$

where f is referred to as the *fractional part* and e as the *exponent*. The fractional part f satisfies $0 \leq f < 1$ and is stored in 52 bits. That means that $2^{52} f$ is an integer in the interval $[0, 2^{52})$. The exponent is restricted to be an integer in the interval $-1022 \leq e \leq 1023$. Instead of storing e, computers store $e + 1023$ using 11 bits. (This obviates the need to store the sign of e.) The sign of x is stored in a separate bit. Thus, the floating-point number x is stored in $52 + 11 + 1 = 64$ bits. Note that the number 1 in the $(1 + f)$ part does not have to be stored explicitly. The binary representation (2) of x is said to be *normalized*, because of the leading 1 in $1 + f$.

Also note that 11 bits can store more integer values than the exponent e is allowed to take on as defined above. The extra values are reserved for some special cases discussed below.

The representable floating-point numbers are equidistant in each interval $2^e \leq x \leq 2^{e+1}$, with distance 2^{e-52}. The constant eps, or *machine epsilon*, is defined to be the distance between 1 and the next largest floating point number. In a relative sense, eps is as large as the gaps between floating-point numbers get. Let the function $\mathrm{fl}(x)$ return the closest floating-point number to the real number x. Then

$$|x - \mathrm{fl}(x)| \leq \frac{\mathsf{eps}}{2}|x|. \tag{3}$$

In IEEE arithmetic, we have $\mathsf{eps} = 2^{-52}$, i.e., $\mathsf{eps} \approx 2.2 \cdot 10^{-16}$. The symbol $\approx$ stands for approximately equal to. You can find out the actual value of eps by first typing format long and then eps in Octave or MATLAB.

The property (3) also can be stated in the form that for every real number x, there is an ϵ with $|\epsilon| \leq \mathsf{eps}/2$, such that

$$\mathrm{fl}(x) = x(1 + \epsilon). \tag{4}$$

Example 2.1

Neither the decimal numbers 0.7 nor 0.1 can be stored exactly as floating-point numbers (2). We have $\mathrm{fl}(0.7)/\mathrm{fl}(0.1) \neq 7$. □

Example 2.2

The largest representable normalized floating point number, realmax, corresponds to $f = 1 - \mathsf{eps}$ and $e = 1023$ in (2), i.e.,

$$\mathsf{realmax} = (2 - \mathsf{eps}) \cdot 2^{1023}.$$

Thus, $\mathsf{realmax} \approx 1.80 \cdot 10^{308}$. □

Real numbers larger than realmax cannot be represented by floating-point numbers. The special "floating-point number" defined by $f = 0$ and $e = 1024$ is called Inf. If in the course of arithmetic operations, a variable, say y, becomes larger than realmax, then y is set to Inf. Replacing a large real number by Inf is referred to as *overflow*.

Inf plus a finite number is defined to be Inf. Moreover, Inf + Inf gives Inf. The expression 1/Inf is defined to be 0. Computations that yield undefined numbers, such as 1/0, 0/0, and Inf/Inf are represented by the special symbol NaN, which stands for "Not a Number." NaN is defined by $f \neq 0$ and $e = 1024$. Arithmetic operations with NaN return NaN. For example, NaN + 1 yields NaN.

The smallest representable positive normalized floating-point number, 2^{-1022}, is denoted by realmin and represented by $f = 0$ and $e = -1022$. Thus, $\mathsf{realmin} \approx 2.2 \cdot 10^{-308}$. If during arithmetic operations an intermediate result, say y, is smaller than realmin, then y is set to zero unless unnormalized floating-point number are used. Replacing a tiny real number by zero is referred to as *underflow*.

Example 2.3

Let $y = 10^{-308}$. First divide y by 10 and then multiply by 10^{308}. On a computer that does not use unnormalized floating-point numbers, we obtain the answer zero due to underflow. This illustrates that underflow may lead to large errors if the intermediate underflown quantity subsequently is multiplied by a large number. □

Some computers allow a representation of numbers between eps · realmin and realmin by denormalized floating-point numbers. Then numbers as small as $0.49 \cdot 10^{-323}$ can be represented. Generally, this feature is not important, but it leads to more accurate answers in certain computations.

Exercise 2.1

Determine the decimal representation of the binary numbers (a) 10, (b) 1011. □

Exercise 2.2

Determine the binary representation of the decimal number (a) 3, (b) 17. □

Exercise 2.3

Determine whether your computer uses unnormalized floating-point numbers. □

Exercise 2.4

If you own a hand-held calculator or have access to one (for example, one of the popular graphing calculators), determine its machine epsilon. □

Many computers also can do arithmetic with single precision floating-point numbers. These numbers are stored in 32 bits each. They require half the computer memory than double precision floating-point numbers. Since they have fewer bits in their mantissa and exponent, computed results may be more affected by round-off errors (defined below), underflow, or overflow, than when double precision floating-point numbers are used. Some of the fastest parallel computers available carry out arithmetic with quadruple precision floating-point numbers. Each such floating-point number is stored in 128 bits. This allows more bits for the mantissa and exponent than in double precision floating-point numbers.

2.2 Floating-Point Arithmetic

Arithmetic operations on floating-point numbers do not always result in another floating-point number. For example, 1 and 10 are floating-point numbers, but 1/10 is not. The result of an arithmetic computation will be stored by the computer as a floating-point number. If the exact result is not a floating-point number, an error is introduced. This error is referred to as *round-off error*.

Let the symbol $*$ represent one of the four basic mathematical arithmetic operations: addition, subtraction, multiplication, or division. Let $\circ$ denote its floating-point analogue. Computers use these operations to carry out most computations. We would like that for any floating-point numbers x and y,

$$x \circ y = \mathrm{fl}(x * y).$$

Combining this property and (4) shows that for all floating-point numbers x and y, there is an ϵ with $|\epsilon| \leq \mathsf{eps}/2$, such that

$$x \circ y = (x * y)(1 + \epsilon). \tag{5}$$

The properties (4) and (5) are satisfied by arithmetic carried out on computer systems that satisfy the IEEE standard, *except* when overflow or underflow occur. The error incurred in (5) also is called a round-off error.

Example 2.4

Computers do not know the functions $\cos(x)$, $\sin(x)$, and e^x for general values of x. Approximations of their values are computed by evaluating rational functions

$$r(x) = p(x)/q(x),$$

where p and q are suitable polynomials. Note that the evaluation of $r(x)$ only requires the four basic arithmetic operations. □

Often the use of floating-point numbers and floating-point arithmetic does not affect the outcome significantly, when compared with applying exact arithmetic on the corresponding real numbers. However, some care should be taken in the design of numerical methods to maintain high accuracy.

Example 2.5

We would like to compute the Euclidean norm of the vector

$$\mathbf{x} = \begin{bmatrix} 10^{160} \\ 10^{100} \end{bmatrix}.$$

Straightforward computation in MATLAB or Octave using the definition of the Euclidean norm

$$\|\mathbf{x}\| = \sqrt{(10^{160})^2 + (10^{100})^2}$$

gives the result `Inf`, because the quantity $\left(10^{160}\right)^2$ cannot be represented in floating-point arithmetic. However, the norm of $\mathbf{x}$ is small enough to represent. We can compute $\|\mathbf{x}\|$ if we avoid the squaring of the first component. This can be achieved by expressing $\mathbf{x}$ as

$$\mathbf{x} = 10^{160} \begin{bmatrix} 1 \\ 10^{-60} \end{bmatrix}.$$

and evaluating

$$\|\mathbf{x}\| = 10^{160}\sqrt{1^2 + (10^{-60})^2}.$$

□

We conclude this lecture with definitions of the absolute and relative errors in an available approximation $\tilde{x}$ of an error-free real quantity x:

$$\textit{absolute error} \text{ in } x = \tilde{x} - x \tag{6}$$

and

$$\textit{relative error} \text{ in } x = \frac{\tilde{x} - x}{|x|}. \tag{7}$$

Example 2.6

The solutions of the quadratic equation

$$x^2 + ax + b = 0 \tag{8}$$

are given by

$$x_{1,2} = \frac{-a \pm \sqrt{a^2 - 4b}}{2}. \tag{9}$$

Let $a = -500000000$ and $b = 1$, and apply MATLAB, using IEEE arithmetic, to compute the roots. We obtain the computed values

```
>> x1=(-a+sqrt(a^2-4*b))/2

x1 =

   500000000

>> x2=(-a-sqrt(a^2-4*b))/2

x2 =

   0
```

It is obvious that zero is not a root of (8). Taylor expansion of the expressions (9) yields

$$\begin{aligned} x_1 &= 5 \cdot 10^8 - 2 \cdot 10^{-9} - 1.6 \cdot 10^{-26} - \dots , \\ x_2 &= \qquad\quad 2 \cdot 10^{-9} + 1.6 \cdot 10^{-26} - \dots . \end{aligned}$$

Hence, the absolute errors in `x1` and `x2` are of about the same magnitude

$$\begin{aligned} \texttt{x1} - x_1 &\approx -2 \cdot 10^{-9}, \\ \texttt{x2} - x_2 &\approx 2 \cdot 10^{-9}, \end{aligned}$$

while the relative errors are of very different magnitudes,

$$\begin{aligned} \frac{\texttt{x1} - x_1}{x_1} &\approx -4 \cdot 10^{-18}, \\ \frac{\texttt{x2} - x_2}{x_2} &\approx -1, \end{aligned}$$

Thus, the larger root is determined with a very small relative error, while the relative error in the computed approximation of the smaller root is very large. The loss of relative accuracy in the smaller root depends on that this root is determined by adding quantities of large magnitude (about $5 \cdot 10^8$) and of opposite sign. The computed approximation `x2` has no correct digit. This effect is referred to as *cancellation of correct digits*.

A more accurate approximation of x_2 can be determined as follows. Observe that

$$x^2 + ax + b = (x - x_1)(x - x_2) = x^2 - (x_1 + x_2)x + x_1 x_2.$$

It follows that $b = x_1 x_2$. Therefore,

$$x_2 = b/x_1. \tag{10}$$

Determining $\mathtt{x}_1$ as described above and then substituting this approximation into (10) yields an approximation `x2` of x_2 of high relative accuracy; see Exercise 2.5. □

Exercise 2.5

Let `x1` be the computed approximation of Example 2.6 of the larger root of equation (8). Compute an approximation of the smaller root by evaluating (10) with x_1 replaced by `x1`. What is the relative error in the so obtained computed approximation of x_2? □

Exercise 2.6

Give an example when the properties (3) and (4) are violated. □

Exercise 2.7

What can we say about the property (5) when (4) is violated? □

Exercise 2.8

Discuss the computation of $f_1(x) = \sqrt{1 - x^2}$ for values of x with $\mathsf{eps} < x < \sqrt{\mathsf{eps}}$. Assume that the computed value of the square-root function satisfies

$$\mathrm{fl}(\sqrt{x}) = \sqrt{x}(1 + \epsilon).$$

Compare the results with those obtained by evaluating $f_2(x) = \sqrt{(1 - x)(1 + x)}$ and $f_3(x) = \sqrt{(1 - x)}\sqrt{(1 + x)}$. □

Exercise 2.9

Consider a computer with 4 digits of precision on which floating-point numbers $x = d_1.d_2d_3d_4 \ldots \cdot 10^e$ are represented by $\text{fl}(x) = d_1.d_2d_3d_4 \cdot 10^e$. Here each d_j and e represent one decimal digit. (a) Determine the computed value of $100.0 + 0.001$ on this computer. (b) The exact solution of the linear system of equations

$$\begin{bmatrix} 0.001 & 100.0 \\ 100.0 & 100.0 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 100.0 \\ 0 \end{bmatrix}$$

is close to $\mathbf{x} = [-1, 1]^T$. Solve this system by Gaussian elimination (without pivoting) on this computer. What is the answer? What is the norm of the error? □

Exercise 2.10

We would like to determine the value of the sum

$$S_\infty = \sum_{j=1}^{\infty} \frac{1}{j^4}$$

and therefore write an Octave or MATLAB program to sum the first terms. Specifically, sum the first n terms in order to compute the partial sum

$$S_n = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \ldots + \frac{1}{n^4},$$

where n is chosen so that $\text{fl}(S_n + \frac{1}{(n+1)^4}) = \text{fl}(S_n)$. What is n (roughly)?

In order to determine whether the computed value S_n is an accurate approximation of S_∞, compute the sum

$$\hat{S}_{2n} = \frac{1}{(2n)^4} + \frac{1}{(2n-1)^4} + \ldots + \frac{1}{2^4} + 1.$$

where summation is carried out in the order indicated. Are $\hat{S}_{2n}$ and S_n the same? If not, then explain why they are not.

Exercise 2.11

Sum the three numbers 1, 10^{20}, and -10^{20} in different ways. First sum $1 + 10^{20} - 10^{20}$ (in order) and then evaluate $10^{20} + 1 - 10^{20}$ (in order). Are the computed sums the same? If not, then explain why the sums are different. □

Exercise 2.12

Discuss how $\cos(x) - 1$ can be evaluated accurately for x-values close to zero. □

Exercise 2.13

How can the norm of an m-vector be computed so that unnecessary overflow is avoided? Use the approach of Example 2.5. Write an m-file that defines a function for the evaluation of vector norm. □