13 Reals

Floating-point numbers and their properties. Pitfalls of numeric computation. Horner's method. Bisection. Newton's method.

13.1 Floating-point numbers

Real numbers, those declared to be of type REAL in a programming language, are represented as floating-point numbers on most computers. A floating-point number z is represented by a (signed) mantissa m and a (signed) exponent e with respect to a base b: $z = \pm\ m \cdot b^{\pm e}$ (e.g., $z = +0.11 \cdot 2^{-1}$). This section presents a very brief introduction to floating-point arithmetic. We recommend [Gol 91] as a comprehensive survey.

Floating-point numbers can only approximate real numbers, and in important ways, they behave differently. The major difference is due to the fact that any floating-point number system is a *finite number system*, as the mantissa m and the exponent e lie in a bounded range. Consider, as a simple example, the following number system:

$z = \pm 0.b_1 b_2 \cdot 2^{\pm e}$, where b_1, b_2, and e may take the values 0 and 1.

The number representation is *not unique*: The same real number may have many different representations, arranged in the following table by numerical value (lines) and constant exponent (columns).

1.5	$+0.11 \cdot 2^{+1}$		
1.0	$+0.10 \cdot 2^{+1}$		
0.75		$+0.11 \cdot 2^{\pm 0}$	
0.5	$+0.01 \cdot 2^{+1}$	$+0.10 \cdot 2^{\pm 0}$	
0.375			$+0.11 \cdot 2^{-1}$
0.25		$+0.01 \cdot 2^{\pm 0}$	$+0.10 \cdot 2^{-1}$
0.125			$+0.01 \cdot 2^{-1}$
0.	$+0.00 \cdot 2^{+1}$	$+0.00 \cdot 2^{\pm 0}$	$+0.00 \cdot 2^{-1}$

The table is symmetric for negative numbers. Notice the cluster of representable numbers around zero. There are only 15 different numbers, but $2^5 = 32$ different representations.

Exercise: A floating-point number system

Consider floating-point numbers represented in a 6-bit "word" as follows: The four bits $b\ b_2\ b_1\ b_0$ represent a signed mantissa, the two bits $e\ e_0$ a signed exponent to the base 2. Every number has the form $x = b\ b_2\ b_1\ b_0 \cdot 2^{\ e\ e_0}$. Both the exponent and the mantissa are integers represented in *2's complement form*. This means that the integer values –2 .. 1 are assigned to the four different representations $e\ e_0$ as shown:

v	e	e_0
0	0	0
1	0	1
–2	1	0
–1	1	1

(a) Complete the following table of the values of the mantissa and their representation, and write down a formula to compute v from $b\ b_2\ b_1\ b_0$.

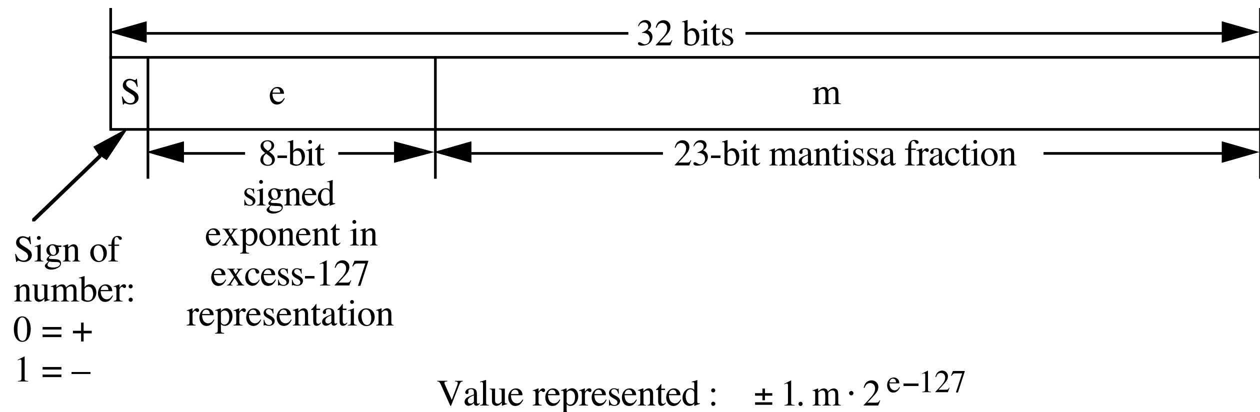
v	b	b_2	b_1	b_0
0	0	0	0	0
1	0	0	0	1
…				
7	0	1	1	1
–8	1	0	0	0
…				
–1	1	1	1	1

(b) How many different number representations are there in this floating-point system?

(c) How many different numbers are there in this system? Draw all of them on an axis, each number with all its representations.

On a byte-oriented machine, floating-point numbers are often represented by 4 bytes = 32 bits: 24 bits for the signed mantissa, 8 bits for the signed exponent. The mantissa m is often interpreted as a fraction $0 \leq m < 1$, whose precision is bounded by 23 bits; the 8-bit exponent permits scaling within the range
$2^{-128} \leq 2^e \leq 2^{127}$. Because 32- and 64-bit floating-point number systems are so common, often coexisting on the same hardware, these number systems are often identified with "single precision" and "double precision", respectively. In recent years an IEEE standard format for-single precision floating-point numbers has emerged, along with standards for higher precisions: double, single extended, and double extended.

IEEE standard single-precision floating-point format:



The following example shows the representation of the number

$$+1.011110 \ldots 0 \cdot 2^{-54}$$

in the IEEE format:

13.2 Some dangers

Floating-point computation is fraught with problems that are hard to analyze and control. Unexpected results abound, as the following examples show. The first two use a binary floating-point number system with a signed 2-bit mantissa and a signed 1-bit exponent. Representable numbers lie in the range

$$-0.11 \cdot 2^{+1} \leq z \leq +0.11 \cdot 2^{+1}.$$

Example: $y + x = y$ and $x \neq 0$

It suffices to choose $|x|$ small as compared to $|y|$; for example,

$$x = 0.01 \cdot 2^{-1}, \; y = 0.10 \cdot 2^{+1}.$$

The addition forces the mantissa of x to be shifted to the right until the exponents are equal (i.e., x is represented as $0.0001 \cdot 2^{+1}$). Even if the sum is computed correctly as $0.1001 \cdot 2^{+1}$ in an accumulator of double length, storing the result in memory will force rounding: $x + y = 0.10 \cdot 2^{+1} = y$.

Example: Addition is not associative: $(x + y) + z \neq x + (y + z)$

The following values for x, y, and z assign different values to the left and right sides.

Left side: $(0.10 \cdot 2^{+1} + 0.10 \cdot 2^{-1}) + 0.10 \cdot 2^{-1} = 0.10 \cdot 2^{+1}$

Right side: $0.10 \cdot 2^{+1} + (0.10 \cdot 2^{-1} + 0.10 \cdot 2^{-1}) = 0.11 \cdot 2^{+1}$

A useful rule of thumb helps prevent the loss of significant digits: Add the small numbers before adding the large ones.

Example: $((x + y)^2 – x^2 – 2xy) / y^2 = 1$?
Let's evaluate this expression for large |x| and small |y| in a floating-point number system with five decimal digits.

$x = 100.00$, $y = .01000$
$x + y = 100.01$
$(x + y)^2 = 10002.0001$, rounded to five digits yields 10002.
$x^2 = 10000.$
$(x + y)^2 – x^2 = 2.????$ (four digits have been lost!)
$2xy = 2.0000$
$(x + y)^2 – x^2 – 2xy = 2.???? – 2.0000 = 0.?????$

Now five digits have been lost, and the result is meaningless.

Example: Numerical instability
Recurrence relations for sequences of numbers are prone to the phenomenon of *numerical instability*. Consider the sequence

$$x_0 = 1.0, \quad x_1 = 0.5, \quad x_{n+1} = 2.5 \cdot x_n – x_{n–1}.$$

We first solve this linear recurrence relation in closed form by trying $x_i = r^i$ for $r \neq 0$. This leads to $r^{n+1} = 2.5 \cdot r^n – r^{n–1}$, and to the quadratic equation
$0 = r^2 – 2.5 \cdot r + 1$, with the two solutions $r = 2$ and $r = 0.5$.

The general solution of the recurrence relation is a linear combination:

$$x_i = a \cdot 2^i + b \cdot 2^{–i}.$$

The starting values $x_0 = 1.0$ and $x_1 = 0.5$ determine the coefficients $a = 0$ and $b = 1$, and thus the sequence is given exactly as $x_i = 2^{–i}$. If the sequence $x_i = 2^{–i}$ is computed by the recurrence relation above in a floating-point number system with one decimal digit, the following may happen:

$x_2 = 2.5 \cdot 0.5 – 1 = 0.2$ (rounding the exact value 0.25),
$x_3 = 2.5 \cdot 0.2 – 0.5 = 0$ (represented exactly with one decimal digit),
$x_4 = 2.5 \cdot 0 – 0.2 = –0.2$ (represented exactly with one decimal digit),
$x_5 = 2.5 \cdot (–0.2) – 0 = –0.5$ (represented exactly with one decimal digit),
$x_6 = 2.5 \cdot (–0.5) – (–0.2) = –1.05$ (exact) $= –1.0$ (rounded),
$x_7 = 2.5 \cdot (–1) – (–0.5) = –2.0$ (represented exactly with one decimal digit),
$x_8 = 2.5 \cdot (–2) – (–1) = –4.0$ (represented exactly with one decimal digit).

As soon as the first rounding error has occurred, the computed sequence changes to the alternative solution $x_i = a \cdot 2^i$, as can be seen from the doubling of consecutive computed values.

Exercise: Floating-point number systems and calculations

(a) Consider a floating-point number system with two ternary digits t_1, t_2 in the mantissa, and a ternary digit e in the exponent to the base 3. Every number in this system has the form $x = .t_1t_2 \cdot 3^e$, where t_1, t_2, and e assume a value chosen among {0, 1, 2}. Draw a diagram that shows all the different numbers in this system, and for each number, all of its representations. How many representations are there? How many different numbers?

(b) Recall the series

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \ldots$$

which holds for $|x| < 1$, for example,

$$\frac{1}{1-\frac{1}{2}} = 2 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots .$$

Use this formula to express 1 / 0.7 as a series of powers.

13.3 Horner's method

A polynomial of n-th degree (e.g., n = 3) is usually represented in the form

$$a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

but is better evaluated in nested form,

$$((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0.$$

The first formula needs n multiplications of the form $a_i \cdot x^i$ and, in addition, n – 1 multiplications to compute the powers of x. The second formula needs only n multiplications in total: The powers of x are obtained for free as a side effect of the coefficient multiplications.

The following procedure assumes that the (n + 1) coefficients a_i are stored in a sufficiently large array a of type 'coeff':

```
type  coeff = array[0 .. m] of real;

function horner(var a: coeff;  n: integer;  x: real): real;
var  i: integer;  h: real;
begin
  h := a[n];
  for  i := n – 1  downto  0  do  h := h · x + a[i];
  return(h)
end;
```

13.4 Bisection

Bisection is an iterative method for solving equations of the form f(x) = 0. Assuming that the function f: R Æ R is continuous in the interval [a, b] and that f(a) · f(b) < 0, a root of the equation f(x) = 0 (a zero of f) must lie in the interval [a, b] (Fig. 13.1). Let m be the midpoint of this interval. If f(m) = 0, m is a root. If f(m) · f(a) < 0, a root must be contained in [a, m], and we proceed with this subinterval; if f(m) · f(b) < 0, we proceed with [m, b]. Thus at each iteration the *interval of uncertainty* that must contain a root is half the size of the interval produced in the previous iteration. We iterate until the interval is smaller than the tolerance within which the root must be determined.

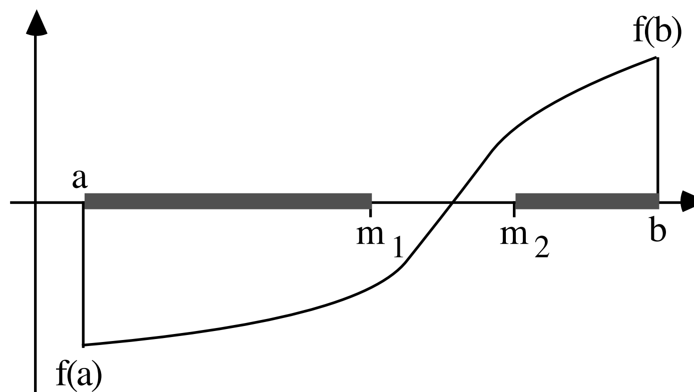


Figure 13.1: As in binary search, bisection excludes half of the interval under consideration at every step.

```
function bisect(function f: real;  a, b: real): real;
const  epsilon = 10^-6;
var  m: real;  faneg: boolean;
begin
  faneg := f(a) < 0.0;
  repeat
    m := (a + b) / 2.0;
    if  (f(m) < 0.0) = faneg  then  a := m  else  b := m
  until  |a – b| < epsilon;
  return(m)
end;
```

A *sequence* $x_1, x_2, x_3, \ldots$ converging to x *converges linearly* if there exist a constant c and an index i_0 such that for all $i > i_0$: $|x_{i+1} - x| \le c \cdot |x_i - x|$. An *algorithm* is said to converge linearly if the sequence of approximations constructed by this algorithm converges linearly. In a linearly convergent algorithm each iteration adds a constant number of significant bits. For example, each iteration of bisection halves the interval of uncertainty in each iteration (i.e., adds one bit of precision to the result). Thus bisection converges linearly with $c = 0.5$. A sequence $x_1, x_2, x_3, \ldots$ *converges quadratically* if there exist a constant c and an index i_0 such that for all $i > i_0$: $|x_{i+1} - x| \le c \cdot |x_i - x|^2$.

13.5 Newton's method for computing the square root

Newton's method for solving equations of the form $f(x) = 0$ is an example of an algorithm with quadratic convergence. Let $f: R \rightarrow R$ be a continuous and differentiable function. An approximation x_{i+1} is obtained from x_i by approximating f(x) in the neighborhood of x_i by its tangent at the point $(x_i, f(x_i))$, and computing the intersection of this tangent with the x-axis (Fig. 13.2). Hence

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

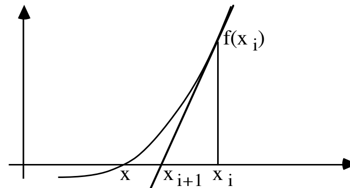


Figure 13.2: Newton's iteration approximates a curve locally by a tangent.

Newton's method is not guaranteed to converge (*Exercise:* construct counterexamples), but when it converges, it does so quadratically and therefore very fast, since each iteration doubles the number of significant bits.

To compute the square root $x = \sqrt{a}$ of a real number $a > 0$ we consider the function $f(x) = x^2 - a$ and solve the equation $x^2 - a = 0$. With $f'(x) = 2 \cdot x$ we obtain the iteration formula:

$$x_{i+1} = x_i - \frac{x_i^2 - a}{2 x_i} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right).$$

The formula that relates x_i and x_{i+1} can be transformed into an analogous formula that determines the propagation of the relative error:

$$R_i = \frac{x_i - x}{x}.$$

Since

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) = \frac{x_i^2 + x^2}{2x_i} = x + \frac{(x_i - x)^2}{2x_i},$$

we obtain for the relative error:

$$\frac{x_{i+1} - x}{x} = \frac{(x_i - x)^2}{2x \cdot x_i} = \frac{1}{2 \cdot \frac{x_i}{x}} \cdot \left(\frac{x_i - x}{x}\right)^2.$$

Using

$$R_i = \frac{x_i - x}{x} \quad \text{and} \quad 1 + R_i = \frac{x_i}{x}$$

we get a recurrence relation for the relative error:

$$R_{i+1} = \frac{R_i^2}{2 \cdot (1 + R_i)}.$$

If we start with $x_0 > 0$, it follows that $1 + R_0 > 0$. Hence we obtain

$$R_1 > R_2 > R_3 > \ldots > 0.$$

As soon as R_i becomes small (i.e., $R_i \ll 1$), we have $1 + R_i \approx 1$, and we obtain

$$R_{i+1} \approx 0.5 \cdot R_i^2.$$

Newton's method converges quadratically as soon as x_i is close enough to the true solution. With a bad initial guess $R_i \gg 1$ we have, on the other hand, $1 + R_i \approx R_i$, and we obtain $R_{i+1} \approx 0.5 \cdot R_i$ (i.e., the computation appears to converge linearly until $R_i \ll 1$ and proper quadratic convergence starts).

Thus it is highly desirable to start with a good initial approximation x_0 and get quadratic convergence right from the beginning. We assume normalized binary floating-point numbers (i.e., $a = m \cdot 2^e$ with $0.5 \le m < 1$). A good approximation of $\sqrt{a}$ is obtained by choosing any mantissa c with $0.5 \le c < 1$ and halving the exponent:

$$x_0 = \begin{cases} c \cdot 2^{e/2} & \text{if } e \text{ is even,} \\ c \cdot 2^{(e+1)/2} & \text{if } e \text{ is odd.} \end{cases}$$

In order to construct this initial approximation x_0, the programmer needs read and write access not only to a "real number" but also to its components, the mantissa and exponent, for example, by procedures such as

```
procedure mantissa(z: real): integer;
procedure exponent(z: real): integer;
procedure buildreal(mant, exp: integer): real;
```

Today's programming languages often lack such facilities, and the programmer is forced to use backdoor tricks to construct a good initial approximation. If x_0 can be constructed by halving the exponent, we obtain the following upper bounds for the relative error:

$$R_1 < 2^{-2}, \quad R_2 < 2^{-5}, \quad R_3 < 2^{-11}, \quad R_4 < 2^{-23}, \quad R_5 < 2^{-47}, \quad R_6 < 2^{-95}.$$

It is remarkable that four iterations suffice to compute an exact square root for 32-bit floating-point numbers, where 23 bits are used for the mantissa, one bit for the sign and eight bits for the exponent, and that six iterations will do for a "number cruncher" with a word length of 64 bits. The starting value x_0 can be further optimized by choosing c carefully. It can be shown that the optimal value of c for computing the square root of a real number is $c = 1 / \sqrt{2} \approx 0.707$.

Exercise: Square root

Consider a floating-point number system with two decimal digits in the mantissa: Every number has the form $x = \pm.d_1\, d_2 \cdot 10^{\pm e}$.

(a) How many different *number representations* are there in this system?

(b) How many different *numbers* are there in this system? Show your reasoning.

(c) Compute $\sqrt{.50 \cdot 10^2}$ in this number system using Newton's method with a starting value $x_0 = 10$. Show every step of the calculation. Round the result of any operation to two digits immediately.

Solution

(a) A number representation contains two sign bits and three decimal digits, hence there are $2^2 \cdot 10^3 = 4000$ distinct number representations in this system.

(b) There are three sources of redundancy:

1. Multiple representations of zero
2. Exponent +0 equals exponent –0
3. Shifted mantissa: $\pm.d0 \cdot 10^{\pm e} = \pm.0d \cdot 10^{\pm e+1}$

A detailed count reveals that there are 3439 different numbers.

Zero has $2^2 \cdot 10 = 40$ representations, all of the form $\pm.00 \cdot 10^{\pm e}$, with two sign bits and one decimal digit e to be freely chosen. Therefore, $r_1 = 39$ must be subtracted from 4000.

If $e = 0$, then $\pm.d_1d_2 \cdot 10^{+0} = \pm.d_1d_2 \cdot 10^{-0}$. We assume furthermore that $d_1d_2 \neq 00$. The case $d_1d_2 = 00$ has been covered above. Then there are $2 \cdot 99$ such pairs. Therefore, $r_2 = 198$ must be subtracted from 4000.

If $d_2 = 0$, then $\pm.d_1 0 \cdot 10^{\pm e} = \pm.0d_1 \cdot 10^{\pm e+1}$. The case $d_1 = 0$ has been treated above. Therefore, we assume that $d_1 \neq 0$. Since $\pm e$ can assume the 18 different values –9, –8, … , –1, +0, +1, … +8, there are $2 \cdot 9 \cdot 18$ such pairs. Therefore, $r_3 = 324$ must be subtracted from 4000.

There are $4000 - r_1 - r_2 - r_3 = 3439$ different numbers in this system.

(c) Computing Newton's square root algorithm:

$$x_0 = 10$$

$$x_1 = .50 \cdot \left(10 + \frac{50}{10}\right) = .50 \cdot (10 + 5) = .50 \cdot 15 = 7.5$$

$$x_2 = .50 \cdot \left(7.5 + \frac{50}{7.5}\right) = .50 \cdot (7.5 + 6.6) = .50 \cdot 14 = 7$$

$$x_3 = .50 \cdot \left(7 + \frac{50}{7}\right) = .50 \cdot (7 + 7.1) = .50 \cdot 14 = 7$$

Exercises

1. Write up all the distinct numbers in the floating-point system with number representations of the form $z = 0.b_1 b_2 \cdot 2^{e_1 e_2}$, where b_1, b_2 and e_1, e_2 may take the values 0 and 1, and mantissa and exponent are represented in 2's complement notation.

2. Provide simple numerical examples to illustrate floating-point arithmetic violations of mathematical identities.