Computational Methods
CMSC/AMSC/MAPL 460

Representing numbers in floating point and associated issues

Ramani Duraiswami,
Dept. of Computer Science

Class Outline

- Computations should be as accurate and as error-free as possible
- Sources of error:
- Poor models of a physical situation
- Ill-posed problems
- Errors due to representation of numbers on a computer and successive operations with these
  - Examples from the book
  - Scientific notation and Floating point representation
  - Concepts: sign, mantissa, base, exponent
  - Distribution of floating point numbers

Modeling

- Original mathematical models may be poorly specified or unavailable
  - E.g. Newton's laws work for non relativistic dynamics
  - Turbulence
  - …
- Computing with a poor model will lead to inevitable errors
- Quantities that are measured may be done so with error and bias
  - Using them in computation will lead to errors
- Approaches to fix these errors are in the domain of interest and statistics
- Not a subject covered in this course

Well posed problems

- Hadamard postulated that for a problem to be “well posed”
  1. *Solution must exist*
  2. *It must be unique*
  3. *Small changes to input data should cause small changes to solution*
- Many problems in science result in “ill-posed” problems.
  - Numerically it is common to have condition 3 violated.
- Converting ill-posed problem to well-posed one is called *regularization*.
- Will discuss this later in the course when talking about optimization and Singular Value Decompositon.

Numerical Modeling and Measurement Errors

- Continuous mathematical models have to be represented in discrete form on the computer
  - Finite-difference or finite-element discretization
  - Continuous quantities may be represented using linear interpolants
  - Model may only reach accurate answer in the limit
  - Round-off errors – continuous numbers represented with discrete representations on the computer
- Focus of today's class:
  - What errors are caused by such representations

Fixed point representation

- How can we represent a number in a computer's memory?
- Fixed point is an obvious way:
- Used to represent integers on computers, and real numbers on some DSPs:
- Each **word** (storage location) in a machine contains a fixed number of digits.
- Example: A machine with a 6-digit word might represent 2005 as

0	0	2	0	0	5

Binary/Decimal/Octal/Hexadecimal

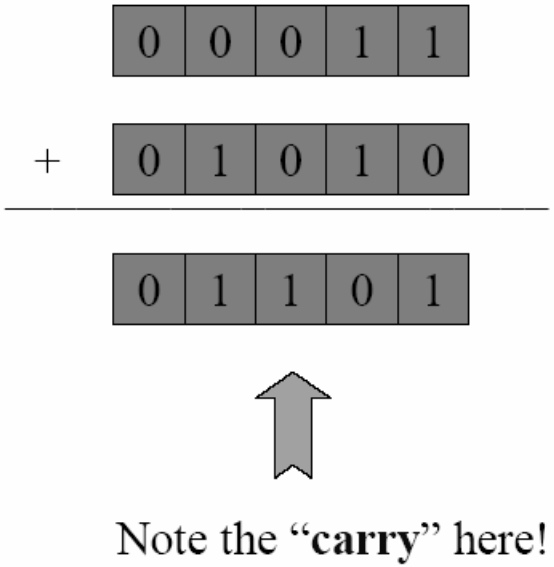
- Numbers can be represented in different bases
- Usually humans use decimal
  - Perhaps because we have ten fingers
- Computer memory often has two states
  - Assigned to 0 and 1
  - Leads to a binary representation
- Octal and Hexadecimal representations arise by considering 3 or 4 memory locations together
  - Lead to 2^3 and 2^4 numbers

Binary Representation

- Most computers use **binary (base 2)** representation.

0 1 0 1 1 0

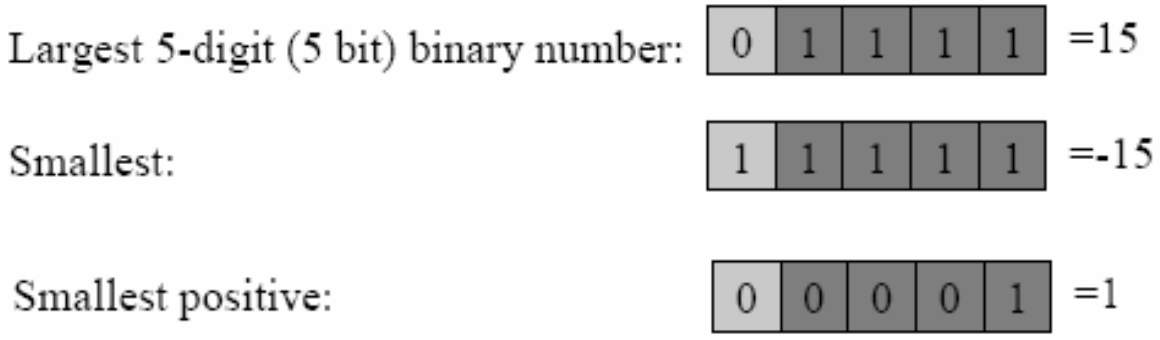
- Each digit has a value 0 or 1.
- If the number above is binary, its value is
- $1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$. (or 22 in base 10)
- Adding numbers in binary



$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 10$ (binary) $= 10_2 = 2$
$1 + 1 + 1 = 11$ (binary) $= 11_2 = 3$

- One way computers represent negative numbers is using the sign-magnitude representation:
- **Sign magnitude** is easiest, and enough for us: if the first bit is zero, then the number is positive. Otherwise, it is negative.
- 0 0 0 1 1 Denotes +11.
- 1 0 0 1 1 Denotes -11.

Range of fixed point numbers



Overflow

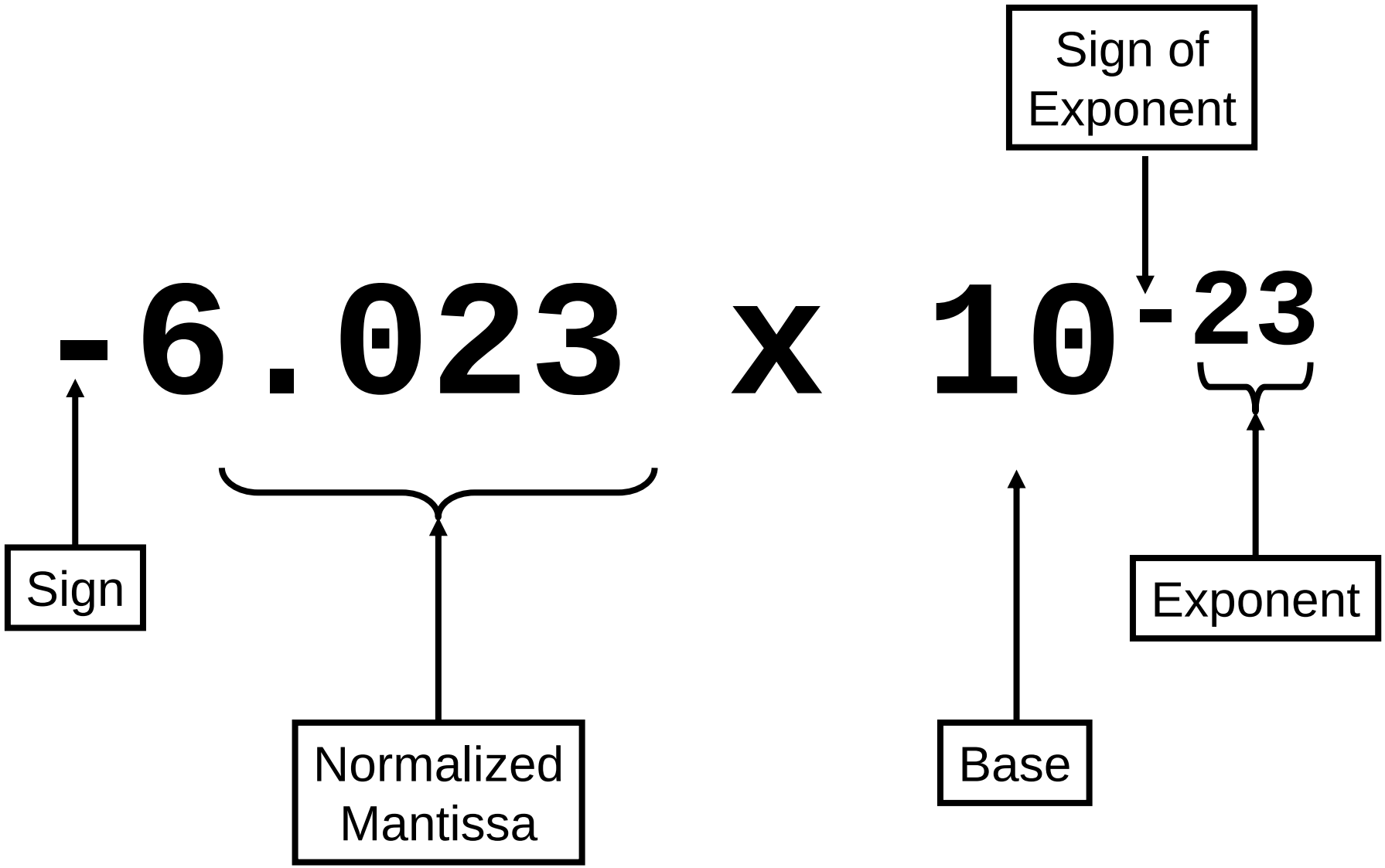
If we try to add these numbers:

	0	1	1	1	1	=15
+	0	1	0	0	0	= 8
we get	1	0	1	1	1	=-7.

We call this "overflow": the answer is too large to store, since it is Outside the range of this number system.

- Fixed point arithmetic:
  - Easy: always get an integer answer.
  - Either we get exactly the right answer, or we can detect
  - overflow.
  - The numbers that we can store are equally spaced.
  - Disadvantage: **very** limited range of numbers.

Scientific Notation



- Once a base is agreed we store each number as two numbers and two signs
  - Mantissa and exponent
- Mantissa is usually “normalized”
- If we have infinite spaces to store these numbers, we can represent arbitrarily large numbers
- With a fixed number of spaces for the two numbers (mantissa and exponent)

Binary Floating Point Representation

- Same basic idea as scientific notation
- Modifications and improvments based on
  - Hardware architecture
  - Efficiency (Space & Time)
  - Additional requirements
    - Infinity
    - Not a number (NaN)
    - Not normalized
    - etc.

Floating point on a computer

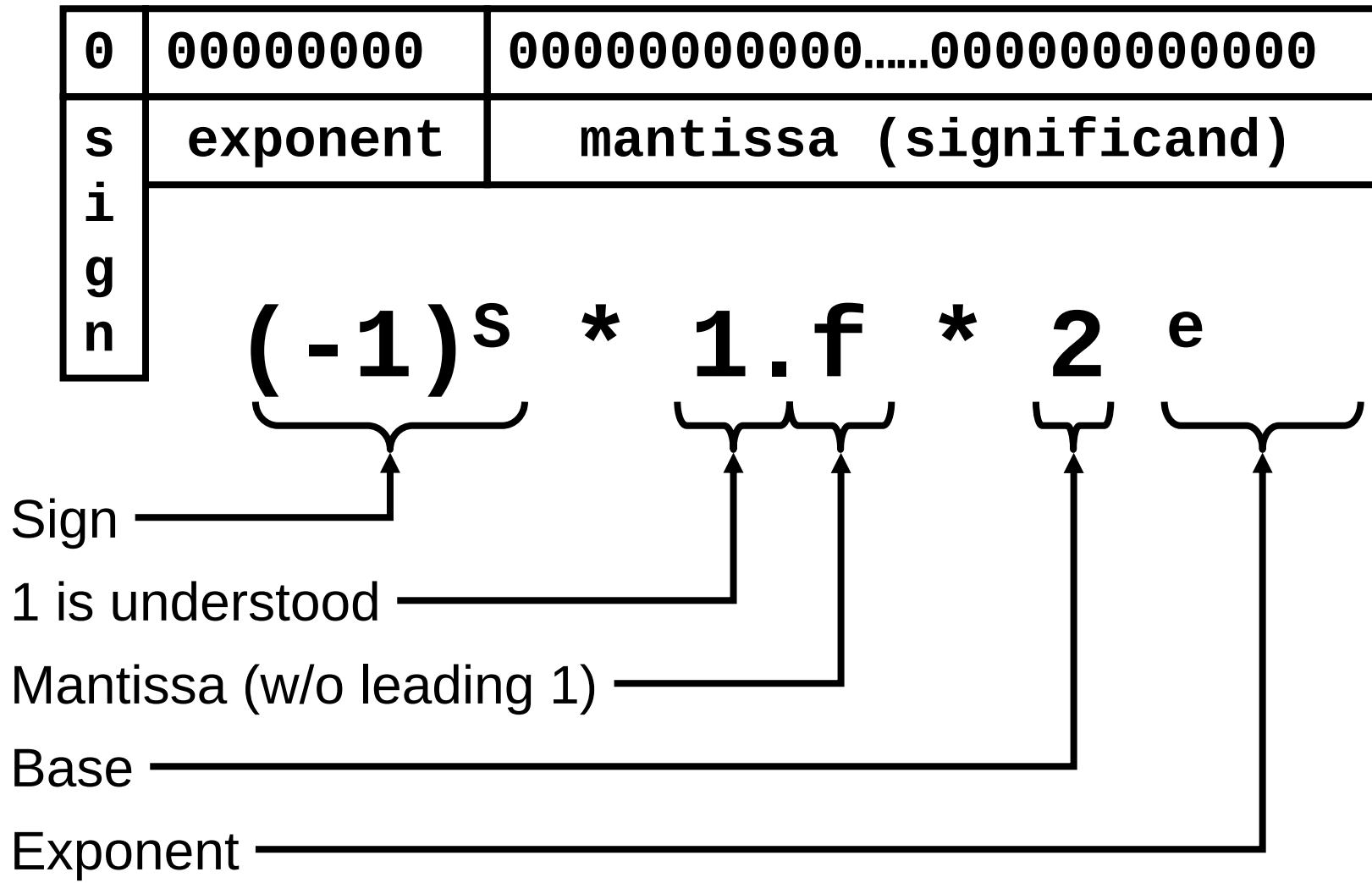
- If we wanted to store 15×2^{11} , we would need 16 bits:

  0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0

- Instead we store it as three numbers
- $(-1)^S \times F \times 2^E$, with F = 15 saved as 01111 and E = 11 saved as 01011.
- Now we can have fractions/decimals, too:

  binary .101 = $1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$.

IEEE-754


0
00000000
00000000000……00000000000
sign
exponent
mantissa (significand)
(-1)S * 1.f * 2 e
Sign
1 is understood
Mantissa (w/o leading 1)
Base
Exponent

IEEE - 754

Most nonzero floating-point numbers are normalized. This means they can be expressed as

$$x = \pm(1 + f) \cdot 2^e$$

The quantity f is the fraction or mantissa and e is the exponent. The fraction satisfies

$$0 \le f < 1$$

and must be representable in binary using at most 52 bits. In other words, $2^{52} f$ is an integer in the interval

$$0 \le 2^{52} f < 2^{52}$$

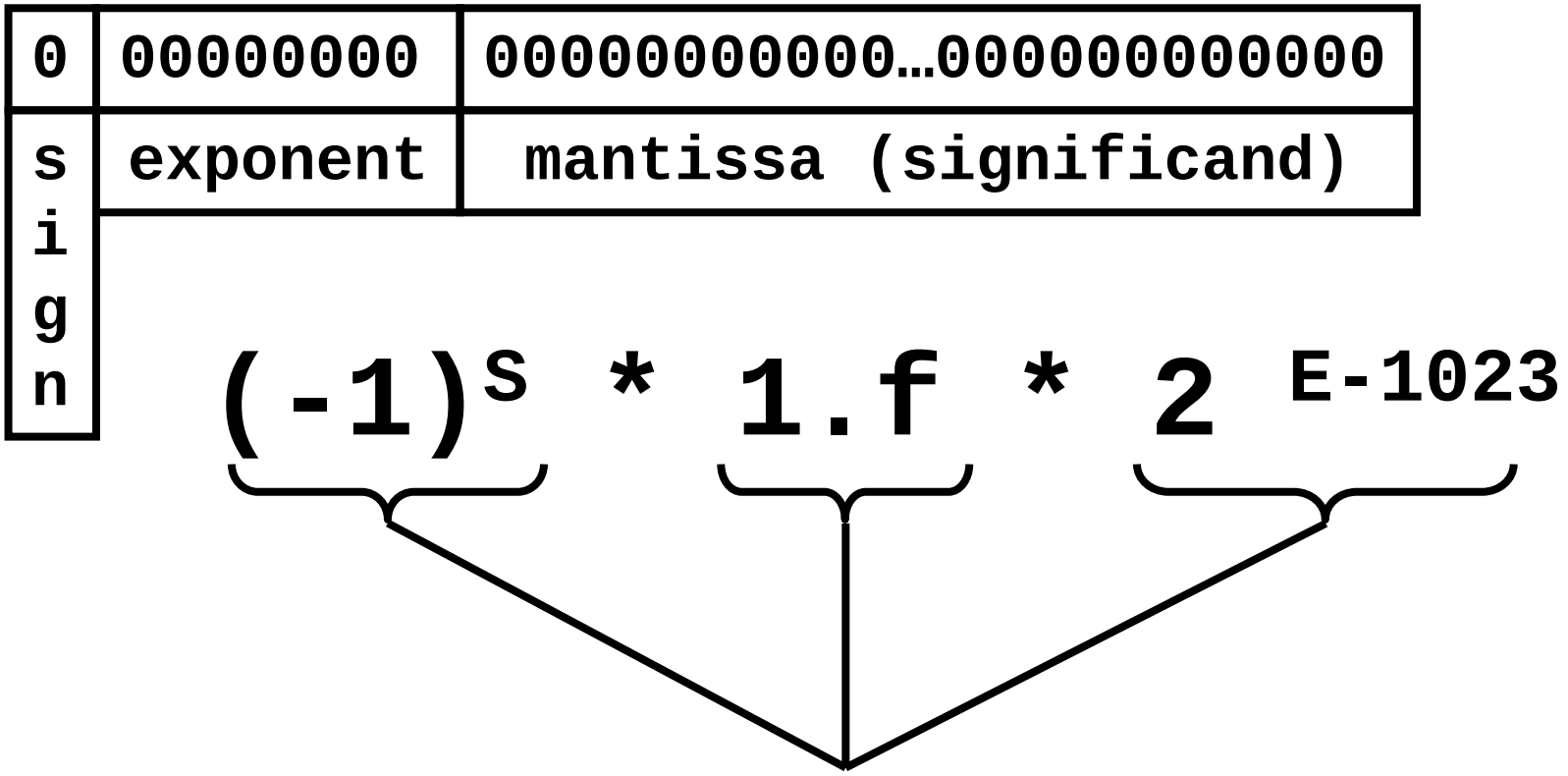
The exponent e is an integer in the interval

$$-1022 \le e \le 1023$$

The finiteness of f is a limitation on *precision*. The finiteness of e is a limitation on *range*. Any numbers that don't meet these limitations must be approximated by ones that do.

Double-precision floating-point numbers are stored in a 64 bit word, with 52 bits for f, 11 bits for e, and one bit for the sign of the number. The sign of e is accommodated by storing $e + 1023$, which is between 1 and $2^{11} - 2$. The two

IEEE-754



Can any of these equal 0?

Can be written...

0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$(-1)^S * 2^E * 1.f$$

	E+1023 == 0	0 < E+1023 < 2047	E+1023 == 2047
f==0	0	Powers of Two	∞
f~=0	Non-normalized typically underflow	Floating point Numbers	Not A Number

- $x = \pm(1+f) \times 2^{e}$
- $0 \leq f < 1$
- $f = (\text{integer} < 2^{52}) / 2^{52}$
- $-1022 \leq e \leq 1023$
- $e = \text{integer}$

Effects of floating point

Finite f implies finite *precision*.

Finite e implies finite *range*

Floating point numbers have discrete spacing,
a maximum and a minimum.

Effects of floating point

- eps *is the distance from* 1 *to the next larger floating-point number.*
- eps = 2^{-52}
- In Matlab

	Binary	Decimal
eps	2^(-52)	2.2204e-16
realmin	2^(-1022)	2.2251e-308
realmax	(2-eps)*2^1023	1.7977e+308

Some numbers cannot be exactly represented

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + \cdots$$

$$t = (1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + \ldots + \frac{9}{16^{12}} + \frac{10}{16^{13}}) \cdot 2^{-4}$$

Problem 1.34.

```
x = 1; while 1+x > 1, x = x/2, pause(.02), end

x = 1; while x+x > x, x = 2*x, pause(.02), end

x = 1; while x+x > x, x = x/2, pause(.02), end
```