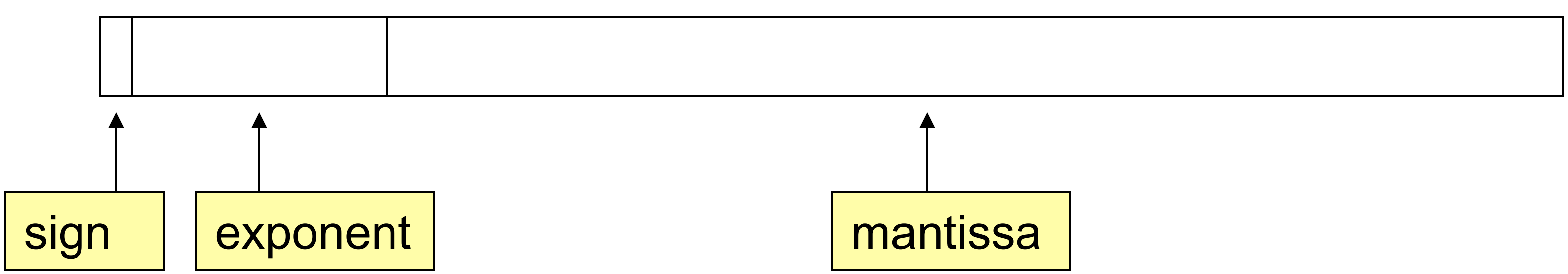
ECS 230

Applied Numerical Linear Algebra

François Gygi
Department of Computer Science

Floating point numbers

- The accessible range of representable numbers can be enhanced by using a mantissa + exponent notation
- $1{,}276{,}448 = 0.1276448 \times 10^7$
- this can be done by dividing a register in 3 fields:



Floating point numbers: the IEEE 754 standard

- Implemented in all modern processors (*)

precision	length	mantissa	exponent	min value	max value
single	32	24	8	1.18×10^{-38}	3.40×10^{38}
double	64	53	11	2.23×10^{-308}	1.79×10^{308}

- single precision: C **`float`**, Fortran **`REAL*4`**
- double precision: C **`double`**, Fortran **`REAL*8`**

(*) GPUs: depends on model

IEEE754 standard

- Exponent: should represent positive and negative exponents
- A bias is added to the actual exponent

  +127 (single precision), +1023 (double precision)
- e = stored value, x = (fraction) * $2^{(e-127)}$
- Allowed values for e: 1 to 254
  - (the values 0 and 255 are reserved for special cases)
- Largest exponent: +127
- Smallest exponent: -126

IEEE754 standard: normalized numbers

- *Normalized* numbers all have a mantissa starting with a 1 (which is therefore not represented)
  - the precision is 24-bit even though 23 bits are used
- How to represent zero?
  - e = mantissa = 0 is a special combination representing zero (all bits in register = 0)
  - Note: +0 and -0 have different representations
- At first sight, single-precision numbers smaller than 1.18×10^{-38} cannot be represented and must be replaced by zero (underflow)

Floating point numbers: denormalized numbers

- If we drop the normalization requirement, numbers smaller than 1.18×10^{-38} can be represented, although with fewer significant digits
- Such numbers are called denormalized numbers
- Denormalized numbers allow for "gradual underflow"

Floating point numbers: denormalized numbers

- Examples of normalized and denormalized numbers (in decimal)
- Assume a minimum exponent of -10, and 4 significant digits
- A normalized number: 1.478×10^{-3}
- Smallest normalized number: 1.000×10^{-10}
- A denormalized number: 0.012×10^{-10}

Floating point numbers

- Smallest number representable with full 24-bit precision: $1.00000\ldots000 \times 2^{-126}$
- FLT_MIN = 1.175e-38 (in header **`float.h`**)
- Smallest number representable

  (with reduced precision, denormalized):

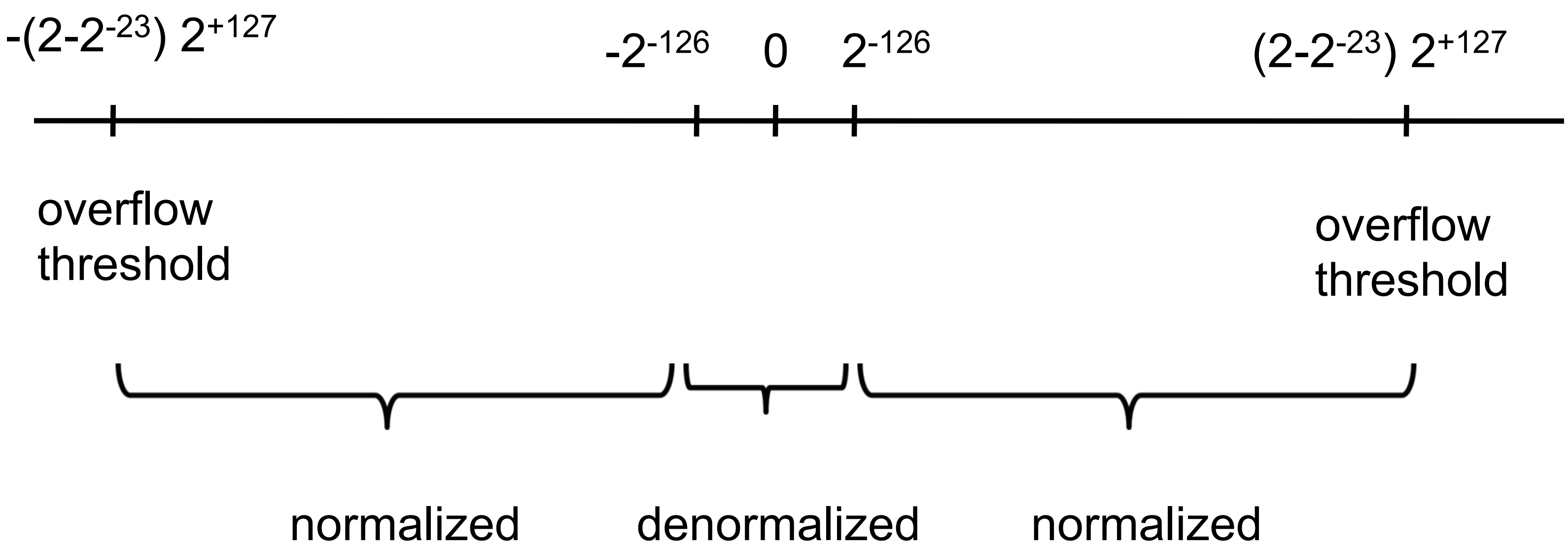
  $0.0000000\ldots00001 \times 2^{-126} = 2^{-149}$
- Largest number representable with full 24-bit precision: $1.11111\ldots111 \times 2^{+127} = (2-2^{-23})\times 2^{+127}$
- FLT_MAX = 3.403e+38

Note: 1/FLT_MAX is not FLT_MIN

fltmax.c

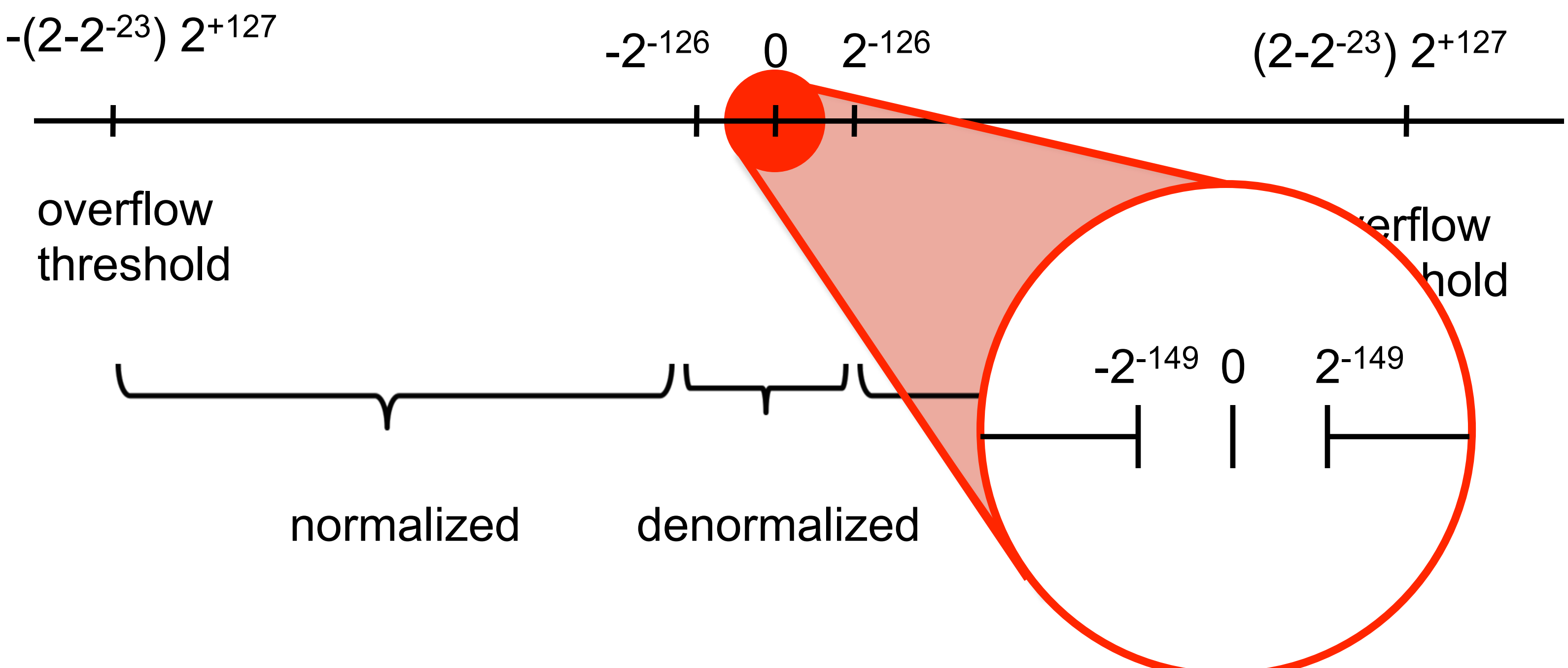
Limitations of floating point

- Ranges of representable numbers



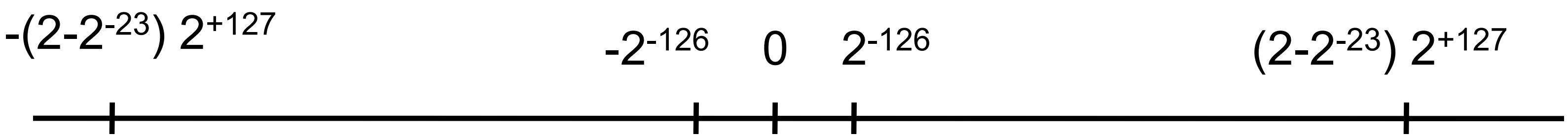
Limitations of floating point

- Ranges of representable numbers



Limitations of floating point

- The density of representable values is irregular



– Example: The nearest larger “neighbor” of

$1.00000000000000000000000 \times 2^{120}$

is

$\underbrace{1.00000000000000000000001}_{\text{24 bits}} \times 2^{120}$

These points are separated by $2^{-23} \times 2^{120} = 2^{97} \sim 10^{29}$

Underflow

- Numbers smaller than 2^{-149} in magnitude are truncated to zero
- $2^{-149} = \approx 1.4013e{-45}$
- The truncation is silent
- Complete loss of information about finiteness of numbers

underflow.c, underflow2.c

Overflow

- Numbers larger than FLT_MAX in magnitude are set to ± Inf
- 1/(+0) = + Inf
- 1/(-0) = - Inf

overflow.c

IEEE 754 special values

- IEEE 754 special values that often (but not always) indicate an incorrect result
  - NaN: “Not a Number” (e=255,fraction≠0)
    - Computing 0/0 yields “NaN”
    - Computing sqrt(-1) yields “NaN”
  - Inf: “Infinity” (e=255,fraction=0)
    - Computing 1/0 yields “Inf”
- Computations that generate “NaN” of “Inf” do not interrupt the program (!)

IEEE 754 special values

- NaN and Inf values “propagate” through other calculations
  - Example: x + Inf = Inf for any x
  - Example: x * NaN = NaN for any x
- Quiet and Signaling NaN values
  - Quiet: does not interrupt calculation
    - (MSB of fraction = 1)
  - Signaling: causes interruption
    - (MSB of fraction = 0)
    - Implementation- and compiler-dependent

fp.c

Most numbers cannot be represented exactly

- But some numbers can..
- Exact representation depends on the base
  - Example: in base 10, 0.1 is exactly representable
  - In floating point binary, 0.1_{10} has a non-terminating representation: $0.00011001\underline{1100}\ldots$
- ½, ¼, etc. are exactly representable in binary floating point
- What is the largest integer that is exactly representable in single precision floating point?

“machine epsilon” or roundoff

- “machine epsilon” is the largest computer-representable number ε such that

$$(1+\varepsilon) - 1 = 0$$

- (machine epsilon also called roundoff *u*)
- How do we determine ε ?

epsilon_f.c, epsilon_d.c

“machine epsilon” or roundoff

- IEEE arithmetic guarantees that

  $|fl(x \text{ op } y) - x \text{ op } y| < \varepsilon \, |x \text{ op } y|$

  for op = +, -, *, /, sqrt

  if no overflow or underflow occurs
- Note: no guarantee about other functions (implemented in libraries): exp, sin, cos, ..
- The order of operations matter

sum3.c

Effects of roundoff

- Cancellation happens when two nearly equal numbers are subtracted
- Example: compute $f(x) = \dfrac{1 - \cos x}{x^2}$

using 10-digit arithmetic

$x = 1.2 \times 10^{-5}$

  - Value of cos(x): c=0.9999999999
  - Subtraction: 1-c = 0.0000000001
  - $(1-c)/x^2 = 10^{-10} / 1.44 \times 10^{-10} = 0.6944$

Effects of roundoff

- $(1-c)/x^2 = 10^{-10} / 1.44 \times 10^{-10} = 0.6944$
- However we know that

$$\cos x = 1 - 2\sin^2(x/2)$$

$$f(x) = \frac{1-\cos x}{x^2} = \frac{1}{2}\left(\frac{\sin(x/2)}{x/2}\right)^2$$

$$f(x) < \frac{1}{2}$$

- The result does not have any correct significant digit

Effects of roundoff

- The subtraction (1-c) is exact, but yields only one correct significant digit
- The result is of the same order of magnitude as the error

Cancellation can lead to a total loss of correct significant digits

Effects of roundoff

- Example: summing a series numerically

$$\boxed{s = \frac{\pi^2}{6} = \sum_{k=1}^{\infty} \frac{1}{k^2}}$$

- All terms positive, no cancellation

float_sum.c

Effects of roundoff

- First approach: sum $1/k^2$ for increasing *k*
- s = 1 + ¼ + 1/9 + 1/25 + 1/36 + 1/49 +…

  result: s = 1.64472532

exact_sum.c

Effects of roundoff

- First approach: sum $1/k^2$ for increasing k
- $s = 1 + ¼ + 1/9 + 1/25 + 1/36 + 1/49 + \ldots$

  result: $s = 1.64472532$
- The correct value is 1.644934066848..
- We have only four correct significant digits (out of possible nine)

Effects of roundoff

- Explanation:
  - at k=4096, the sum is ~1.6, and

  $1/k^2 = 4096^{-2} = 2^{-24}$
  - Single precision has a 24-bit mantissa
  - The contribution from the term k=4096 “drops off the end”: it is too small compared to 1.6, as ε is too small compared to 1
  - All further terms beyond k=4096 do not contribute to the sum

Effects of roundoff

- Solution: sum the series starting with small terms first
  - Note: this requires knowing how many terms to take before the summation begins

float_sum2.c

Effects of roundoff

- Solution: sum the series starting with small terms first
  - Note: this requires knowing how many terms to take before the summation begins
- Using 10^9 terms, and starting from the smallest term, we get 1.64493406 (correct to eight significant digits)

float_sum2.c

Effects of roundoff

- It is not always possible to know in advance what values in a sum are small
  – Example: compute a scalar product of two vectors *x* and *y* in N dimensions

$$s = \sum_{k=1}^{N} x_k y_k$$

- Conclusion: computing sums using a 9-digit mantissa does not guarantee 9 correct significant digits