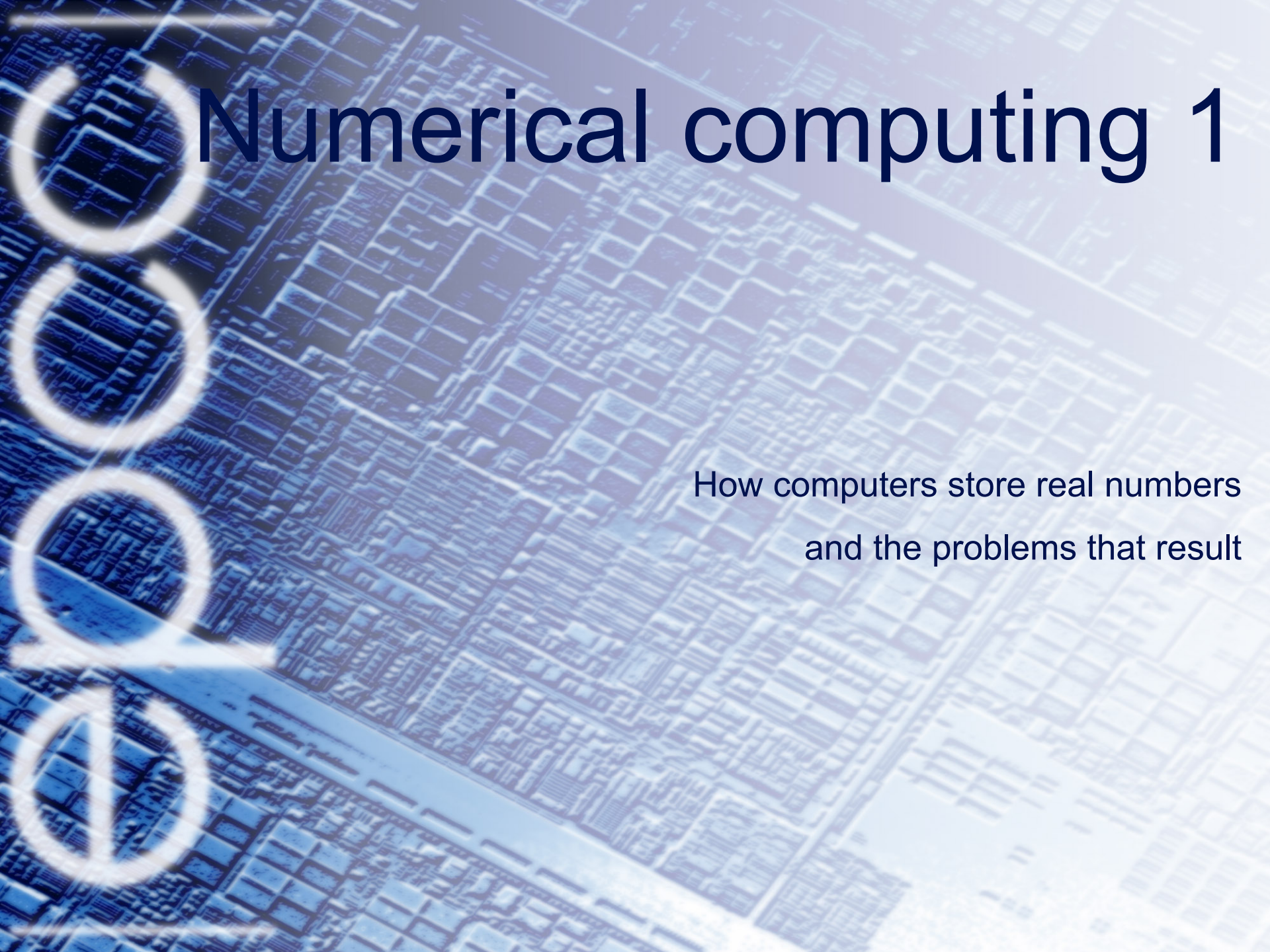

Numerical computing 1
How computers store real numbers
and the problems that result

Overview

- Integers
- Reals, floats, doubles, etc.
- Arithmetical operations and rounding errors

- We write:

```
x = sqrt(2.0)
```

  - but how is this stored?

Mathematics vs Computers

- Mathematics is an ideal world
  - integers can be as large as you want
  - real numbers can be as large or as small as you want
  - can represent every number exactly:

$$1, -3, 1/3, 10^{36237}, 10^{-232322}, \sqrt{2}, \pi, ...$$

- Numbers range from - ∞ to +∞
  - there is also infinite numbers in any interval
- This not true on a computer
  - numbers have a limited range (integers and real numbers)
  - limited precision (real numbers)

Integers

- We like to use
  - we only write the 10 characters 0,1,2,3,4,5,6,7,8,9
  - use *position* to represent each power of 10

$$\mathbf{125} = \mathbf{1} * 10^2 + \mathbf{2} * 10^1 + \mathbf{5} * 10^0$$
$$= \mathbf{1}*100 + \mathbf{2}*10 + \mathbf{5}*1 = \mathbf{125}$$

  - represent positive or negative using a leading “+” or “-”
- Computers are binary machines
  - can only store ones and zeros
  - minimum storage unit is 8 bits = 1 byte
- Use

$$\mathbf{1111101} = \mathbf{1}*2^6 + \mathbf{1}*2^5 + \mathbf{1}*2^4 + \mathbf{1}*2^3 + \mathbf{1}*2^2 + \mathbf{0}*2^1 + \mathbf{1}*2^0$$
$$= \mathbf{1}*64 + \mathbf{1}*32 + \mathbf{1}*16 + \mathbf{1}*8 + \mathbf{1}*4 + \mathbf{0}*2 + \mathbf{1}*1$$
$$= \mathbf{125}$$

Storage and Range

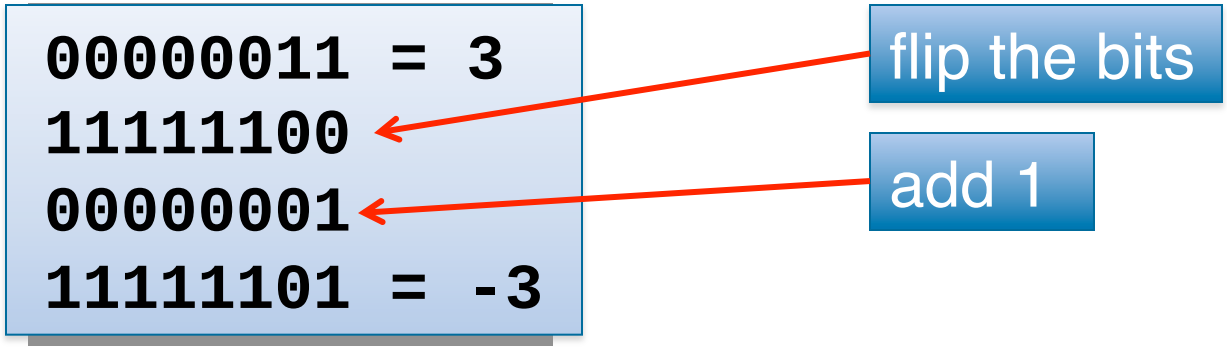
- Assume we reserve 1 byte (8 bits) for integers
  - minimum value **0**
  - maximum value **$2^8 - 1 = 255$**
  - if result is out of range we will overflow and get wrong answer!

- Standard storage is 4 bytes = 32 bits
  - minimum value **0**
  - maximum value **$2^{32} - 1 = 4294967291$ = 4 billion = 4G**
- Is this a problem?
  - question: what is a 32-bit operating system?

- Can use 8 bytes (64 bit integers)

Aside: Negative Integers

- Use “two’s complement” representation
  - flip all ones to zeros and zeros to ones
  - then add one (ignoring overflow)
- Negative integers have the first bit set to “1”
  - for 8 bits, range is now: -128 to + 127
  - normal addition (ignoring overflow) gives the correct answer




```
125 + (-3) = 01111101 + 111111101 = 01111010  = 122
```

Integer Arithmetic

- Computers are brilliant at integer maths
- These can be added, subtracted and multiplied with complete accuracy…
  - …as long as the final result is not too large in magnitude
- But what about division?
  - 4/2 = 2, 27/3 = 9, but 7/3 = 2 (instead of 2.333333333333…).
  - what do we do with numbers like that?
  - how do we store real numbers?

Fixed-point Arithmetic

- Can use an integer to represent a real number.
  - we have 8 bits stored in X 0-255.
  - represent real number *a* between 0.0 and 1.0 by dividing by 256
  - e.g. *a* = 5/9 = 0.55555 represented as X=142
    - 142/256 = 0.5546875
  - X = integer($a \times 256$), Y=integer($b \times 256$), Z=integer(c x 256) ....

- Operations now treat integers as fractions:
  - E.g. c = a × b becomes 256c = (256a × 256b)/256,

    i.e.Z = X×Y/256
  - Between the upper and lower limits (0.0 & 1.0), we have a uniform grid of possible 'real' numbers.

Problems with Fixed Point

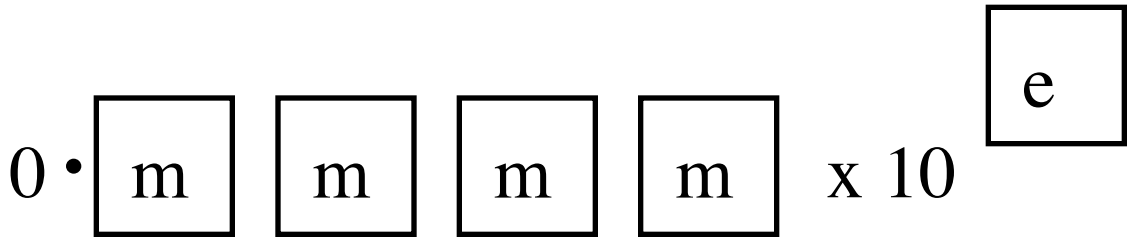
- This arithmetic is very fast
  - but does not cope with large ranges
  - eg above, cannot represent numbers < 0 or numbers >= 1
- Can adjust the range
  - but at the cost of precision

Scientific Notation (in Decimal)

- How do we store 4261700.0 and 0.042617
  - in the same storage scheme?
- Decimal point was previously *fixed*
  - now let it *float* as appropriate
- Shift the decimal place so that it is at the start
  - ie 0.42617 (this is the mantissa *m*)
- Remember how many places we have to shift
  - ie +7 or -1 (the exponent *e*)
- Actual number is $0.mmmm \times 10^{e}$
  - ie $0.4262 * 10^{+7}$ or $0.4262 * 10^{-1}$
    - always use all 5 numbers - don't waste space storing leading zero!
    - automatically adjusts to the magnitude of the number being stored
    - could have chosen to use 2 spaces for *e* to cope with very large numbers

Decimal floating point – worked example

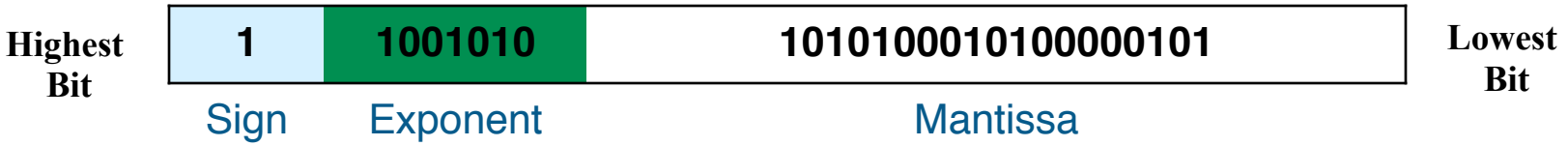
Floating-Point Numbers



- Decimal point “floats” left and right as required
  - fixed-point numbers have constant absolute error, eg +/- 0.00001
  - floating-point have a constant relative error, eg +/- 0.001%
- Computer storage of real numbers directly analogous to scientific notation
  - except using binary representation not decimal
  - ... with a few subtleties regarding sign of *m* and *e*
- All modern processors are designed to deal with floating-point numbers *directly in hardware*

The IEEE 754 Standard

- Mantissa made positive or negative:
  - the first bit indicates the sign: 0 = positive and 1 = negative.
- General binary format is:



- Exponent made positive or negative using a "biased" or "shifted" representation:
  - If the stored exponent, c, is X bits long, then the actual exponent is c – *bias* where the offset *bias* = $(2^X/2 - 1)$. e.g. X=3:

Stored (c,binary)	000	001	010	011	100	101	110	111
Stored (c,decimal)	0	1	2	3	4	5	6	7
Represents (c-3)	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**

IEEE – The Hidden Bit

- In base 10 exponent-mantissa notation:
  - we chose to standardise the mantissa so that it always lies in the binary range $0.0 \leq m < 1.0$
  - the first digit is always 0, so there is no need to write it.
- The FP mantissa is "normalised" to lie in the range:

$$1.0 \leq m < 10.0 \qquad \text{ie decimal range } [1.0,2.0)$$

  - as the first bit is always one, there is no need to store it, We only store the variable part, called the significand (f).
  - the mantissa m = 1.f (in binary), and the 1 is called "The Hidden Bit":
  - however, this means that zero requires special treatment.
    - having f and e as all zeros is defined to be (+/-) zero.

Binary Fractions: what does 1.f mean?

- Whole numbers are straightforward
  - base 10: $109 = 1*10^2 + 0*10^1 + 9*10^0 = 1*100 + 0*10 + 9*1 = 109$
  - base 2: $1101101 = 1*2^6+1*2^5+0*2^4+1*2^3+1*2^2+0*2^1+1*2^0$
    $= 1*64 + 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1$
    $= 64 + 32 + 8 + 4 + 1 = 109$
- Simple extension to fractions

$109.625 = 1*10^2 + 0*10^1 + 9*10^0 + 6*10^{-1} + 2*10^{-2} + 5*10^{-3}$
$= 1*100 + 0*10 + 9*1 + 6*0.1 + 2*0.01 + 5*0.001$

$1101101.101 = 109 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$
$= 109 + 1*(1/2) + 0*(1/4) + 1*(1/8)$
$= 109 + 0.5 + 0.125$
$= 109.625$

Relation to Fixed Point

- Like fixed point with divisor of 2^n
  - base 10: $109.625 = 109 + 625 / 10^3 = 109 + (625 / 1000)$
  - base 2: $1101101.101 = 1101101 + (101 / 1000)$
    $= 109 + 5/8 = 109.625$

- Or can think of shifting the decimal point

  $109.625 = 109625/10^3 = 109625 / 1000$ (decimal)

  $1101101.101 = 1101101101 / 1000$ (binary)

  $= 877/8 = 109.625$

IEEE – Bitwise Storage Size

- The number of bits for the mantissa and exponent.
  - The normal floating-point types are defined as:

Type	Sign, a	Exponent, c	Mantissa, f	Representation
Single 32bit	1bit	8bits	23+1bits	$(-1)^s \times 1.f \times 2^{c-127}$ **Decimal: ~8s.f. × $10^{\sim\pm38}$**
Double 64bit	1bit	11bits	52+1bits	$(-1)^s \times 1.f \times 2^{c-1023}$ **Decimal: ~16s.f. × $10^{\sim\pm308}$**

  - there are also “Extended” versions of both the single and double types, allowing even more bits to be used.
  - the Extended types are not supported uniformly over a wide range of platforms; Single and Double are.

32-bit and 64-bit floating point

- Conventionally called single and double precision
  - C, C++ and Java: **`float`** (32-bit), **`double`** (64-bit)
  - Fortran: **`REAL`** (32-bit), **`DOUBLE PRECISION`** (64-bit)
    - or **`REAL(KIND(1.0e0))`**, **`REAL(KIND(1.0d0))`**
    - or **`REAL (Kind=4), REAL (Kind=8)`**
  -
- Single precision accurate to 8 significant figures
  - eg 3.2037743 E+03
- Double precision to 16
  - eg 3.203774283170437 E+03
- Fortran usually knows this when printing default format
  - C and Java often don't
  - depends on compiler

Summary

- Real numbers stored in floating-point format
- Conform to IEEE 754 standard
  - defines storage format
  - can be single (32-bit) and double (64-bit) precison
  - and the result of all arithmetical operations
- Lots of issues to be aware of… see you next time