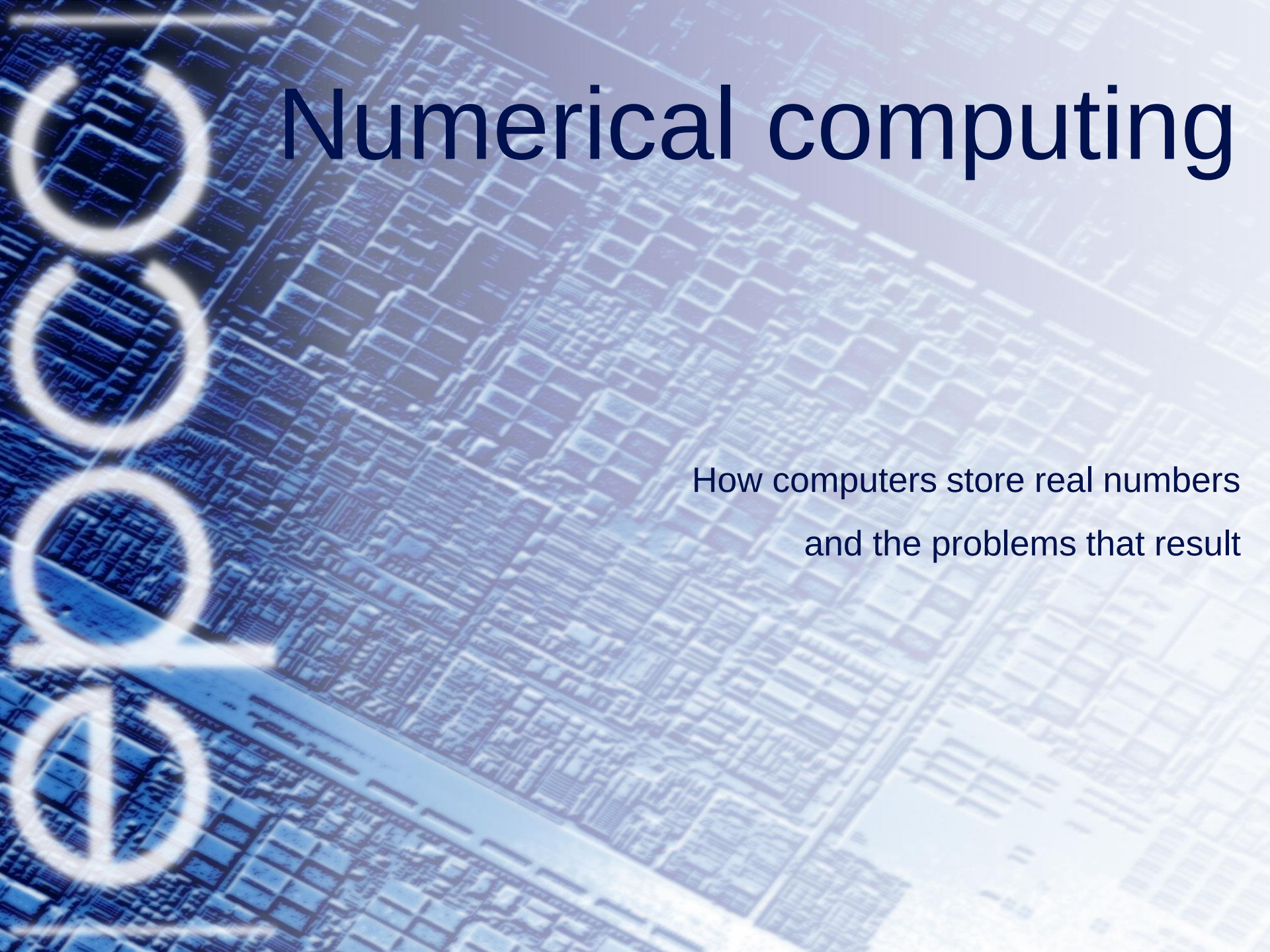

Numerical computing
How computers store real numbers
and the problems that result

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 **base 10**
  - 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 **base 2**

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

Storage and Range

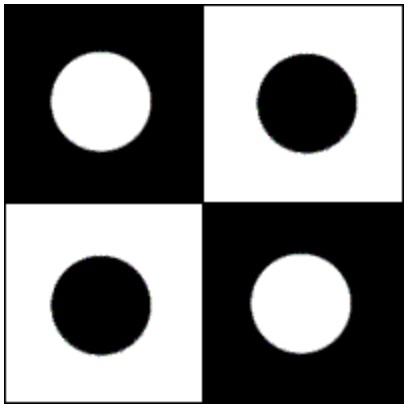
- Assume we reserve 1 byte (8 bits) for integers
  - minimum value `0`
  - maximum value $2^8 - 1 = 255$
  - if result is out of range we will overflow and get wrong answer!

- 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



```
00000011 = 3
11111100
00000001
11111101 = -3
```

flip the bits

add 1

```
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.

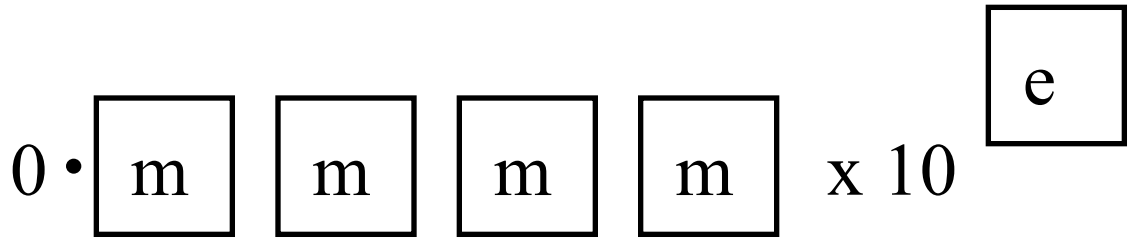
Problems with Fixed Point

- 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

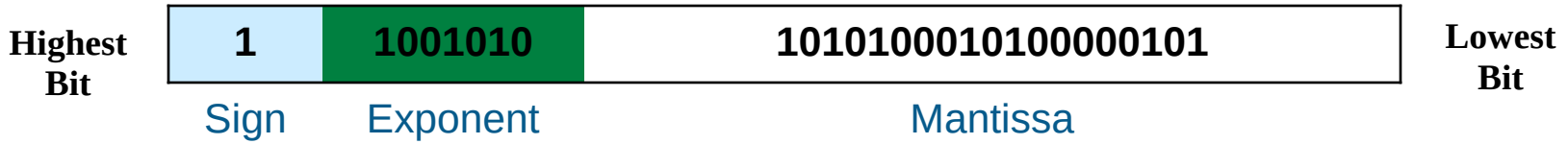
Floating-Point Numbers



- 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 **binary** 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. × $10^{\sim\pm 38}$**
Double 64bit	1bit	11bits	52+1bits	$(-1)^s \times 1.f \times 2^{c-1023}$ **Decimal: ~16s.f. × $10^{\sim\pm 308}$**

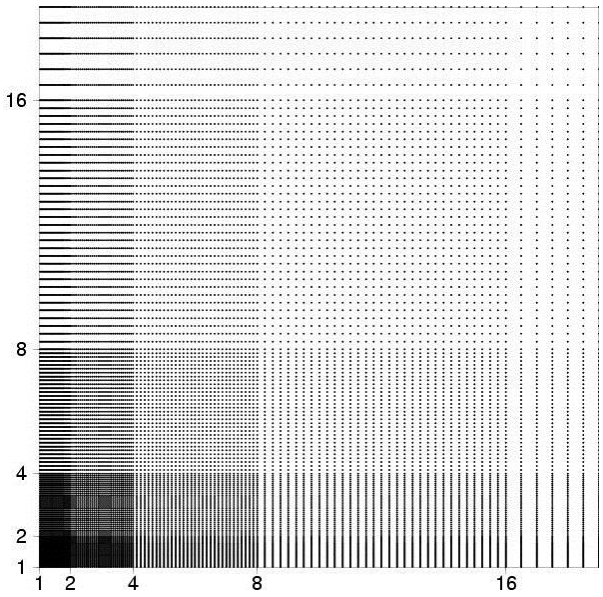
  - there are also “Extended” versions of both the single and double types, allowing even more bits to be used.
  - the Extended types are not supported uniformly over a wide range of platforms; Single and Double are.

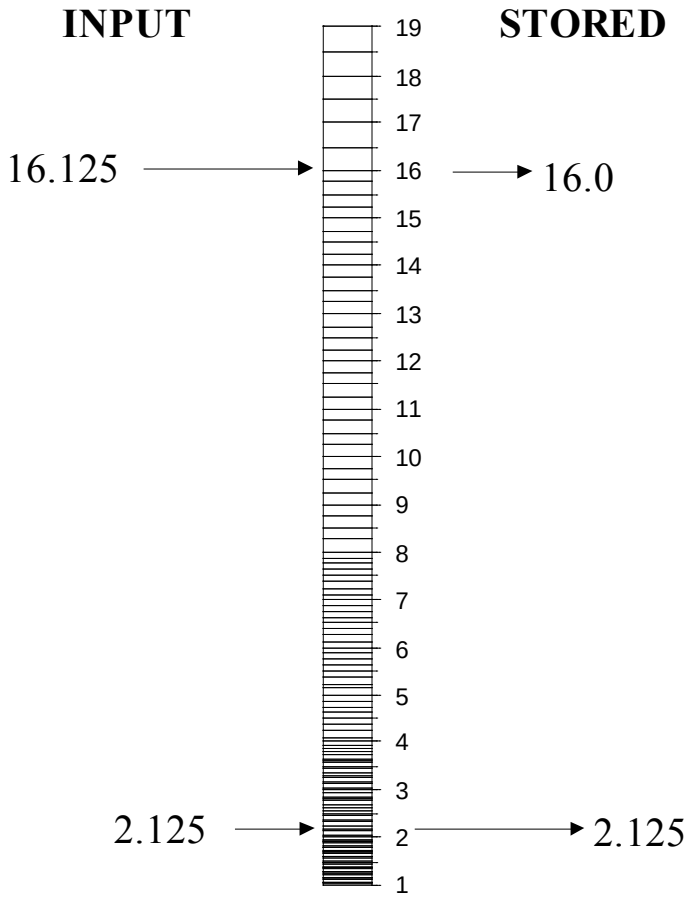
32-bit and 64-bit floating point

- 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)`**
  - **NOTHING TO DO with 32-bit / 64-bit operating systems!!!**
- 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

IEEE Floating-point Discretisation

- This still cannot represent all numbers:
- And in two dimensions you get something like:





Limitations

- Numbers cannot be stored exactly
  - gives problems when they have very different magnitudes
- Eg 1.0E-6 and 1.0E+6
  - no problem storing each number separately, but when adding:

    0.000001 + 1000000.0 = 1000000.000001 = 1.000000000001E6

  - in 32-bit will be rounded to 1.0E6
- So

    (0.000001 + 1000000.0) - 1000000.0 = 0.0

    0.000001 + (1000000.0 - 1000000.0) = 0.000001

  - FP arithmetic is commutative but not associative!

Example I

```
Program quad
  implicit none

  real (kind=16) :: q23
  real (kind=8)  :: d23
  real (kind=4)  :: s23

  integer :: ctr

  s23=2.0/3.0
  d23=2.0_8/3.0_8
  q23=2.0_16/3.0_16

  do ctr = 1,24
     s23 = s23/10.0 + 1
     d23 = d23/10.0 + 1
     q23 = q23/10.0 + 1
     write(*,*) s23, d23, q23
  end do

  do ctr = 1,24
     s23 = (s23 - 1)*10.0
     d23 = (d23 - 1)*10.0
     q23 = (q23 - 1)*10.0
     write(*,*) s23, d23, q23
  end do

end Program quad
```

start with ⅔
single, double, quadruple
divide by 10 add 1
repeat many times (18)
subtract 1 multiply by 10
repeat many times (18)

What happens to ⅔ ?

The output

1.0666667	1.0666666666666667	1.0666666666666666666666666666666667
1.1066667	1.1066666666666667	1.1066666666666666666666666666666667
1.1106666	1.1106666666666667	1.1106666666666666666666666666666666
1.1110667	1.1110666666666666	1.1110666666666666666666666666666666
1.1111066	1.1111066666666667	1.1111066666666666666666666666666666
1.1111107	1.1111106666666666	1.1111106666666666666666666666666667
1.1111110	1.1111110666666666	1.1111110666666666666666666666666667
1.1111112	1.1111111066666666	1.1111111066666666666666666666666667
1.1111112	1.1111111106666667	1.1111111106666666666666666666666666
1.1111112	1.1111111110666667	1.1111111110666666666666666666666667
1.1111112	1.1111111111066667	1.1111111111066666666666666666666667
1.1111112	1.1111111111106666	1.1111111111066666666666666666666668
1.1111112	1.1111111111110668	1.1111111111106666666666666666666667
1.1111112	1.1111111111111067	1.1111111111110666666666666666666667
1.1111112	1.1111111111111107	1.1111111111111066666666666666666666
1.1111112	1.1111111111111112	1.1111111111111106666666666666666666
1.1111112	1.1111111111111112	1.1111111111111110666666666666666667
1.1111112	1.1111111111111112	1.1111111111111111066666666666666666
1.1111116	1.1111111111111116	1.1111111111111111066666666666666663
1.1111164	1.1111111111111160	1.1111111111111111066666666666666635
1.1111641	1.1111111111111605	1.1111111111111111066666666666666349
1.1116409	1.1111111111116045	1.1111111111111110666666666666663487
1.1164093	1.1111111111160454	1.1111111111111066666666666666634870
1.1640930	1.1111111111604544	1.1111111111110666666666666666348700
1.6409302	1.1111111116045436	1.1111111111066666666666666663487003
6.4093018	1.1111111160454357	1.1111111110666666666666666634870034
54.093018	1.1111111604543567	1.1111111106666666666666666348700338
530.93018	1.1111116045435665	1.1111111066666666666666663487003375
5299.3018	1.1111160454356650	1.1111110666666666666666634870033754
52983.016	1.1111604543566500	1.1111106666666666666666348700337535
529820.13	1.1116045435665001	1.1111066666666666666663487003375350
5298191.0	1.1160454356650007	1.1110666666666666666634870033753501
52981900.	1.1604543566500070	1.1106666666666666666348700337535014
5.29819008E+08	1.6045435665000696	1.1066666666666666663487003375350137
5.29819034E+09	6.0454356650006957	1.0666666666666666634870033753501367
5.29819034E+10	50.454356650006957	0.6666666666666663487003375350136689

The result: Two thirds

Single precision
fifty three billion !

Double precision
fifty!

Quadruple precision
has no information about two-thirds after 18^{th} decimal place

Example II – order matters!

```c
#include <stdio.h>

int main()
{
  float a,b,c,x,y;
  double da,db,dc,dx,dy;

  a = -1.0e10;
  b = 1.0e10;
  c = 1.0;

  x = a + b;
  x = x + c;

  y = b + c;
  y = a + y;

  da = -1.0e10;
  db = 1.0e10;
  dc = 1.0;

  dx = da + db;
  dx = dx + dc;

  dy = db + dc;
  dy = da + dy;

  printf("float:x=%f y=%f\n",x,y);
  printf("double:x=%f y=%f\n",dx,dy);


  return 0;
}
```

This code adds three numbers together in a different order. Single and double precision.

$$x = \left(-1.0 \times 10^{10} + 1.0 \times 10^{10}\right) + 1.0$$

$$y = -1.0 \times 10^{10} + \left(1.0 \times 10^{10} + 1.0\right)$$

What is the answer?

The result. One

```
float:x=1.000000 y=0.000000
double:x=1.000000 y=1.000000
```

Example III: Gauss

- C. 1785AD in what is now Lower Saxony, Germany
  - School teacher sets class a problem
  - Sum numbers 1 to 100
  - Nine year old boy quickly has the answer

$$S_n = \sum_{i=1}^{n} i = \frac{n}{2}(n+1)$$

$$S_{100} = \frac{100}{2}(100+1) = 5050$$

Carl Friedrich Gauss
(C.1840 AD)

Summing numbers

```c
#include <stdio.h>

int main()
{

  int i,m;
  float sumup,sumdown;
  int n=100;

  for(m=0;m<3;m++){

    sumup=0;
    for(i=0;i<=n;i++){
      sumup+=(float)i;
    }
    sumdown=0;
    for(i=n;i>=1;i--){
      sumdown+=(float)i;
    }
    printf("gauss:%d %f %f %d\n",n, sumup, sumdown,n*(n+1)/2);
    n*=10;
  }

  return 0;
}
```

sums numbers to 100, 1000, 10000
performs sum low-to-high and high-to-low in single precision

The result:Gauss' sum

```
gauss:100 5050.000000 5050.000000 5050
gauss:1000 500500.000000 500500.000000 500500
gauss:10000 50002896.000000 50009072.000000 50005000
```

In single precision summing numbers 1 to 10000 produces the wrong answer
high-to-low and low-to-high produce different wrong answers

What happens when in parallel
same calculation, different numbers of processors!

Special Values

- We have seen that zero is treated specially
  - corresponds to all bits being zero (except the sign bit)
- There are other special numbers
  - infinity: which is usually printed as “Inf”
  - Not a Number: which is usually printed as “NaN”
- These also have special bit patterns

Infinity and Not a Number

- Infinity is usually generated by dividing any finite number by 0.
  - although can also be due to numbers being too large to store
  - some operations using infinity are well defined, e.g. -3/ ∞ = -0
- NaN is generated under a number of conditions:

$$\infty + (- \infty), \quad 0 \times \infty, \quad 0/0, \quad \infty/\infty, \quad \sqrt{(X)} \text{ where } X < 0.0$$

  - most common is the last one, eg x = sqrt(-1.0)
- Any computation involving NaN's returns NaN.
  - there is actually a whole set of NaN binary patterns, which can be used to indicate why the NaN occurred.

IEEE Special Values

Exponent, e (unshifted)	Mantissa, f	Represents
000000…	0	±0
000000…	≠0	$0.f \times 2^{(1-bias)}$ [denormal]
000… < e < 111…	Any	$1.f \times 2^{(e-bias)}$
111111…	0	±∞
111111…	≠0	NaN

- Most numbers are in standard form (middle row)
  - have already covered zero, infinity and NaN
  - but what are these “denormal numbers” ???

Range of Single Precision

- Have 8 bits for exponent, 1+23 bits for mantissa
  - unshifted exponent can range from 0 to 255 (bias is 127)
  - smallest and largest values are reserved for denormal (see later) and infinity or NaN
  - unshifted range is 1 to 254, shifted is -126 to 127
- Largest number:

$$1.11111111111111111111111 \times 2^{127}$$

$$\sim 2 \times 2^{127} = 2^{128} \sim \mathbf{3.4 \times 10^{38}}$$

- Smallest number

$$1.00000000000000000000000 \times 2^{-126}$$

$$= 2^{-126} \sim \mathbf{1.2 \times 10^{-38}}$$

- But what is smallest exponent reserved for ...?

IEEE Denormal Numbers

- Standard IEEE has mantissa normalised to 1.xxx
- But, normalised numbers can give x-y=0 when x≠y!
  - consider 1.10×2^{-Emin} and 1.00×2^{-Emin} where *Emin* is smallest exponent
  - upon subtraction, we are left with 0.10×2^{-Emin}.
  - in normalised form we get $1.00 \times 2^{-Emin-1}$:
    - this cannot be stored because the exponent is too small.
    - when normalised it must be flushed to zero.
  - thus, we have $x \neq y$ while at the same time $x-y = 0$!
- Thus, the smallest exponent is set aside for *denormal* numbers, beginning with 0.f (not 1.f).
  - can store numbers smaller than the normal minimum value
    - but with reduced precision in the mantissa
  - ensures that x = y when x-y = 0 (also called *gradual underflow*)

Denormal Example

- Consider the single precision bit patterns:
  - mantissa: 0000100....
  - exponent: 00000000
- Exponent is zero but mantissa is non-zero
  - a denormal number
  - value is 0. 0000100... x 2^{-126} ~ 2^{-5} x 2^{-126} = 2^{-131} ~ 3.7E-40
- Smaller than normal minimum value
  - but we lose precision due to all the leading zeroes
  - smallest possible number is 2^{-23} x 2^{-126} = 2^{-149} ~ 1.4E-45

Exceptions

- May want to terminate calculation if any special values occur
  - could indicate an error in your code
- Can usually be controlled by your compiler
  - default behaviour can vary
  - eg some systems terminate on NaN, some continue
- Usual action is to terminate and dump the core

IEEE Arithmetic Exceptions

Exception	Result
Overflow	$\pm\infty$, f = 11111…
Underflow	0, $\pm 2^{-bias}$, [denormal]
Divide by zero	$\pm\infty$
Invalid	NaN
Inexact	*round*(x)

- It is not necessary to catch all of these.
  - inexact occurs extremely frequently and is usually ignored
  - underflow is also usually ignored
  - you probably want to catch the others

IEEE Rounding

- We wish to add, subtract, multiply and divide.
  - E.g. Addition of two 3d.p. decimal numbers:

0.1241×10^{-1}	+	0.2815×10^{-2}	=	
0.1241×10^{-1}	+	0.02815×10^{-1}	=	0.15225×10^{-1}
But can only store 4 decimal places:				0.1522×10^{-1} or 0.1523×10^{-1}

- In essence:
  - we shift the decimal (radix) point,
  - perform fixed point arithmetic,
  - renormalise the number by shifting the radix point again.
- But what do we do with that 5?
  - do we round up, round down, truncate, ...

IEEE Rounding Modes

- Rounding types:
  - there are four types of rounding for arithmetic operations.
    - Round to nearest: e.g. -0.001298 becomes -0.00130.
    - Round to zero: e.g. -0.001298 becomes -0.00129.
    - Round to +infinity: e.g. -0.001298 becomes -0.00129.
    - Round to –infinity: e.g. -0.001298 becomes -0.00130.
  - but how can we ensure the rounding is done correctly?
- Guard digits:
  - calculations are performed at slightly greater precision on the CPU, and then stored in standard IEEE floating-point numbers.
  - usually uses three extra binary digits to ensure correctness.
- Your compiler may be able to change the mode

Implementations: C & FORTRAN

- Most C and FORTRAN compilers are fully IEEE 754 compliant.
  - compiler switches are used to switch on exception handlers.
  - these may be very expensive if dealt with in software.
  - you may wish to switch them on for testing (except inexact), and switch them off for production runs.
- But there are more subtle differences.
  - FORTRAN always preserves the order of calculations:
    - A + B + C = (A + B) +C, always.
  - C compilers are free to modify the order during optimisation.
    - A + B + C may become (A + B) + C or A + (B + C).
    - Usually, switching off optimisations retains the order of operations.

Implementations: Java

- In summary:
  - Java only supports round-to-nearest.
  - Java does not allow users to catch floating-point exceptions.
  - Java only has one NaN.
- All of this is technically a bad thing
  - these tools can be used to to test for instabilities in algorithms
  - this is why Java does not support these tools, and also why hardcore numerical scientists don't like Java very much
  - however, Java also has some advantages over, say, C
    - forces explicit casting
    - you can use the strictfp modifier to ensure that the same bytecode produces identical results across all platforms.

Summary

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

- All real calculations suffer from rounding errors
  - important to choose an algorithm where these are minimised

- Practical exercise illustrates the key points