Numerical computing 2

How computers store real numbers

and the problems that result

The IEEE 754 Standard

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

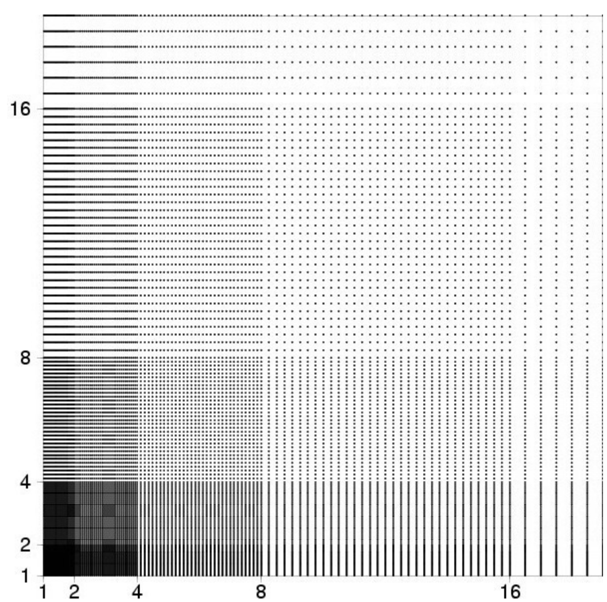
Highest Bit	1	1001010	1010100010100000101	Lowest Bit
	Sign	Exponent	Mantissa	

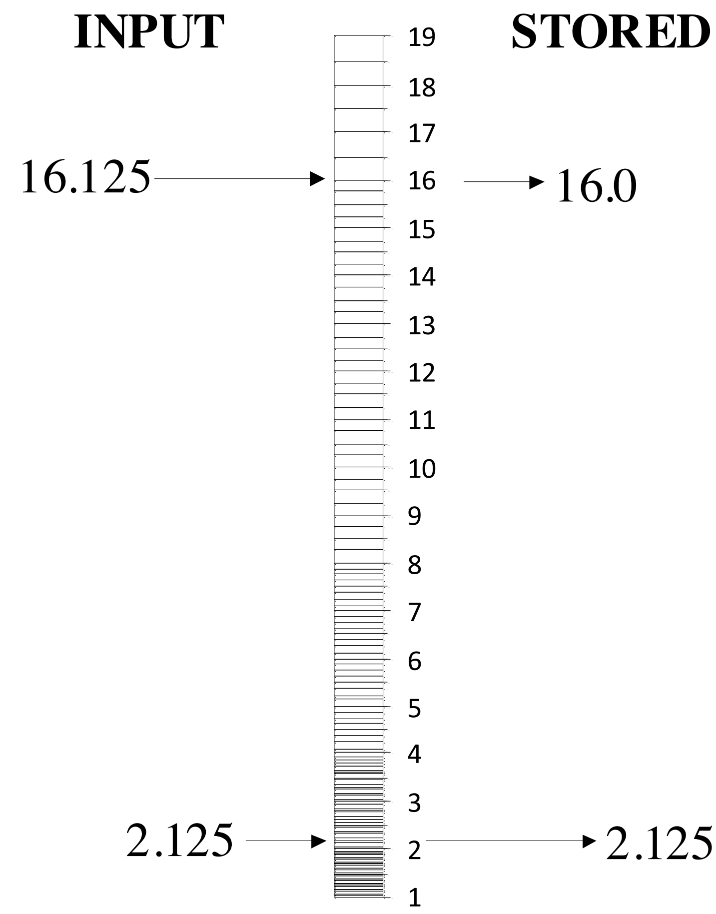
- Exponent made positive or negative using a “biased” or “shifted” representation:
  - If the stored exponent, c, is X bits long, then the actual exponent is c – *bias* where the offset *bias* = $(2^X/2 - 1)$. e.g. X=3:

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 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 recurrence
  implicit none
  real (kind=16) :: q23
  real (kind=8) :: d23
  real (kind=4) :: s23
  integer :: i

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

  do i = 1,18
     s23 = s23 / 10 + 1
     d23 = d23 / 10 + 1
     q23 = q23 / 10 + 1
     write(*,*) s23, d23, q23
  end do

  do i = 1,18
     s23 = (s23 - 1) * 10
     d23 = (d23 - 1) * 10
     q23 = (q23 - 1) * 10
     write(*,*) s23, d23, q23
  end do

end program recurrence
```

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

The output

```
$ gfortran recurrence.f90 -o recurrence
$ ./recurrence
   1.06666672       1.0666666666666667       1.06666666666666666666666666666666665
   1.10666668       1.1066666666666667       1.10666666666666666666666666666666668
   1.11066663       1.1106666666666667       1.11066666666666666666666666666666661
   1.11106670       1.1110666666666666       1.11106666666666666666666666666666660
   1.11110663       1.1111066666666667       1.11110666666666666666666666666666664
   1.11111069       1.1111106666666666       1.11111066666666666666666666666666666
   1.11111104       1.1111110666666666       1.11111106666666666666666666666666672
   1.11111116       1.1111111066666666       1.11111110666666666666666666666666669
   1.11111116       1.1111111106666667       1.11111111066666666666666666666666663
   1.11111116       1.1111111110666667       1.11111111106666666666666666666666672
   1.11111116       1.1111111111066667       1.11111111110666666666666666666666675
   1.11111116       1.1111111111106666       1.11111111111066666666666666666666675
   1.11111116       1.1111111111110668       1.11111111111106666666666666666666671
   1.11111116       1.1111111111111067       1.11111111111110666666666666666666667
   1.11111116       1.1111111111111107       1.11111111111111066666666666666666665
   1.11111116       1.1111111111111112       1.11111111111111106666666666666666657
   1.11111116       1.1111111111111112       1.11111111111111110666666666666666673
   1.11111116       1.1111111111111112       1.11111111111111111066666666666666663
   1.11111164       1.1111111111111116       1.11111111111111110666666666666666635
   1.11111641       1.1111111111111160       1.11111111111111106666666666666666349
   1.11116409       1.1111111111111605       1.11111111111111066666666666666663487
   1.11164093       1.1111111111116045       1.11111111111110666666666666666634870
   1.11640930       1.1111111111160454       1.11111111111106666666666666666348700
   1.16409302       1.1111111111604544       1.11111111111066666666666666663487003
   1.64093018       1.1111111116045436       1.11111111110666666666666666634870034
   6.40930176       1.1111111160454357       1.11111111106666666666666666348700338
   54.0930176       1.1111111604543567       1.11111111066666666666666663487003375
   530.930176       1.1111116045435665       1.11111110666666666666666634870033754
   5299.30176       1.1111160454356650       1.11111106666666666666666348700337535
   52983.0156       1.1111604543566500       1.11111066666666666666663487003375350
   529820.125       1.1116045435665001       1.11110666666666666666634870033753501
   5298191.00       1.1160454356650007       1.11106666666666666666348700337535014
   52981900.0       1.1604543566500070       1.11066666666666666663487003375350137
   529819008.       1.6045435665000696       1.10666666666666666634870033753501367
   5.29819034E+09   6.0454356650006957       1.06666666666666666348700337535013669
   5.29819034E+10   50.454356650006957       0.666666666666666634870033753501366885
```

The result: Two thirds

Single precision
fifty three billion !

Double precision
fifty!

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

Example II – order matters!

```
#include <iostream>

template <typename T>
void order(const char* name) {
  T a, b, c, x, y;

  a = -1.0e10;
  b = 1.0e10;
  c = 1.0;
  x = (a + b) + c;
  y = a + (b + c);

  std::cout << name << ": x = " << x << ", y = " << y << std::endl;
}
int main()
{
  order<float>(" float");
  order<double>("double");
  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

```
$ clang++ -O0 order.cpp -o order
$ ./order
 float: x = 1, y = 0
double: x = 1, y = 1
```

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 sum_up, sum_down;
  int n = 100;

  for (m = 0; m < 3; ++m) {
    sum_up = 0;
    for (i = 1; i <= n; ++i) {
      sum_up += i;
    }

    sum_down = 0;
    for (i = n; i >= 1; --i) {
      sum_down += i;
    }

    printf("Gaussian sum up to %5d: %11.1f %11.1f %9.d\n",
            n, sum_up, sum_down, n*(n+1)/2);
    n *= 10;
  }
}
```

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

The result:Gauss' sum

```
$ clang gauss.c -o gauss
$ ./gauss
Gaussian sum up to   100:      5050.0      5050.0      5050
Gaussian sum up to  1000:    500500.0    500500.0    500500
Gaussian sum up to 10000:  50002896.0  50009072.0  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. $-3/\infty = -0$

- 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 $x \neq y$!
  - 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 $2^{-126} \sim 2^{-5} \times 2^{-126} = 2^{-131} \sim 3.7E-40$
- Smaller than normal minimum value
  - but we lose precision due to all the leading zeroes
  - smallest possible number is $2^{-23} \times 2^{-126} = 2^{-149} \sim 1.4E-45$

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

- Floating point numbers defined in IEEE 754 standard

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

- Practical exercise illustrates the key points