Representing Numbers of a Computer, Part 2

How computers store real numbers
and the problems that result

1

IEEE – Bitwise Storage Size

Highest Bit	**1**	**10010101**	**10000010000011101000100**	**Lowest Bit**
	Sign	Exponent	Mantissa	

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

Type	Sign, s	Exponent, c	Mantissa, f	Representation
Single 32-bit	1bit	8bits	23+1 bits	$(-1)^s \times 1.f \times 2^{c-127}$ **Decimal: ~8s.f. × $10^{\sim\pm 38}$**
Double 64-bit	1bit	11bits	52+1 bits	$(-1)^s \times 1.f \times 2^{c-1023}$ **Decimal: ~16s.f. × $10^{\sim\pm 308}$**

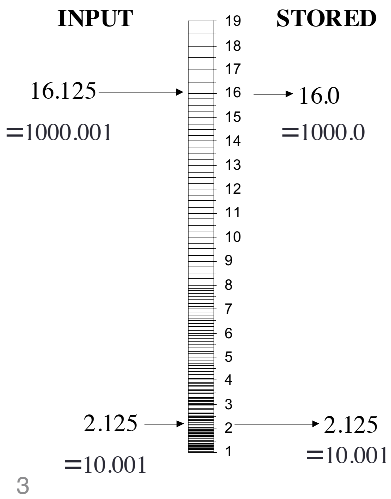
  – there are also “Extended” versions of both the single and double types, allowing even more bits to be used. “Half Precision” (16-bit) also available for graphics, etc.
  – the Extended types are not supported uniformly over a wide range of platforms; Single and Double are.

2

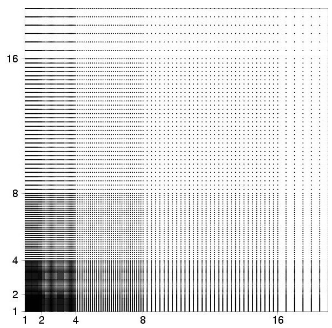
2

IEEE Floating-point Discretisation

- This still cannot represent all numbers.
- E.g. with 5 bits for mantissa (including hidden bit):



- and in two dimensions
  you get something like:



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
  - E.g. 3.2037743E+03
- Double precision accurate to ~16 significant figures
  - E.g. 3.203774283170437E+03
- Fortran usually knows this when printing default format
  - C and Java often don't
  - depends on compiler

Limitations

- Numbers cannot be stored exactly
  - gives problems when they have very different magnitudes
- E.g. 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: A + B = B + A
  - …but not in general associative (A + B) + C ≠ A + (B + C)

epcc

5

5

Example I

```
program recurrence
  implicit none
  real (kind=4) :: s23
  real (kind=8) :: d23
  real (kind=16) :: q23
  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
- Set up single, double, quadruple
- divide by 10 add 1
- repeat many times (18)
- subtract 1 multiply by 10
- repeat many times (18)

6

6

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         1.11111111111111111106666666666666635
   1.11111641       1.1111111111111160         1.11111111111111111066666666666666349
   1.11116409       1.1111111111111605         1.11111111111111110666666666666663487
   1.11164093       1.1111111111116045         1.11111111111111106666666666666634870
   1.11640930       1.1111111111160454         1.11111111111111066666666666666348700
   1.16409302       1.1111111111604544         1.11111111111110666666666666663487003
   1.64093018       1.1111111116045436         1.11111111111106666666666666634870034
   6.40930176       1.1111111160454357         1.11111111110666666666666666348700338
   54.0930176       1.1111111604543567         1.11111111106666666666666663487003375
   530.930176       1.1111116045435665         1.11111111066666666666666634870033754
   5299.30176       1.1111160454356650         1.11111110666666666666666348700337535
   52983.0156       1.1111604543566500         1.11111106666666666666663487003375350
   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         0.66666666666666663487003375350136688
```


The result: Two thirds

Single precision
fifty three billion!

Double precision
fifty!

Quadruple precision loses 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?

epcc 9

9

The result. One

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

epcc 10

10

Example III: Gauss

- C. 1785 AD 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

```
#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

- In floating point numbers, zero is treated specially
  - corresponds to all bits being zero (except the sign bit)
    - Zero can have a sign
    - Both $+0.0$ and -0.0 equate to be the same in calculations
    - $1.0/(-0.0) = -\infty$ and $1.0/(+0.0) = +\infty$
- 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:

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

  - most common is the last one, e.g. $x = \text{sqrt}(-1.0)$
- Any computation involving NaNs returns NaN.
  - there is actually a whole set of NaN binary patterns, which can be used to indicate why the NaN occurred.

epcc 15

15

IEEE Special Values

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

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

epcc 16

16

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 zero, subnormal (see later) and infinity or NaN
  - unshifted range is then 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 the case of the zero exponent (non-zero mantissa) reserved for ...?

epcc 17

17

IEEE Subnormal 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, giving a gap in this region.
  - thus, we have $x \neq y$ while at the same time $x - y = 0$!
- Thus, the smallest exponent is set aside for *subnormal* (or *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*)

epcc 18

18

Subnormal Example

- Consider the single precision bit patterns:
  - mantissa: 0000100....
  - exponent: 00000000
- Exponent is zero but mantissa is non-zero
  - a subnormal 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

epcc 19

19

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
  - E.g. some systems terminate on NaN, some continue
- Usual action is to terminate and dump the core

epcc 20

20

IEEE Arithmetic Exceptions

Exception	Result
Overflow	$\pm\infty$, f = 11111…
Underflow	0, $\pm 2^{-bias}$, [subnormal]
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

epcc 21

21

IEEE Rounding

- We wish to add, subtract, multiply and divide.
  - E.g. Addition of two decimal numbers, given to 4 s.f.:

0.1241×10^{-1}	+	0.2815×10^{-2}	=	
0.1241×10^{-1}	+	0.02815×10^{-1}	=	$0.1522\mathbf{5}\times10^{-1}$
But can only store 4 significant figures:				0.1522×10^{-1} or 0.1523×10^{-1}

- In essence:
  - we shift the decimal point on one input as required,
  - perform fixed point arithmetic,
  - renormalise the number by shifting the decimal point again.
- But what do we do with that 5?
  - do we round up, round down, truncate, ...

epcc 22

22

IEEE Rounding Modes

- We can choose from several 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 bits to ensure correctness.
  - Guarantees an FP operation gives correct result up to rounding
    - answer will still be inaccurate due to rounding and rounding errors will accumulate
- Your compiler may be able to change the mode
  - Round to nearest is default

epcc 23

23

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.
- Complex numbers usually stored as pair of floating point numbers
  - C (ISO C99) and Fortran support for going between real and complex

epcc 24

24

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 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.

epcc 25

25

Other Precisions

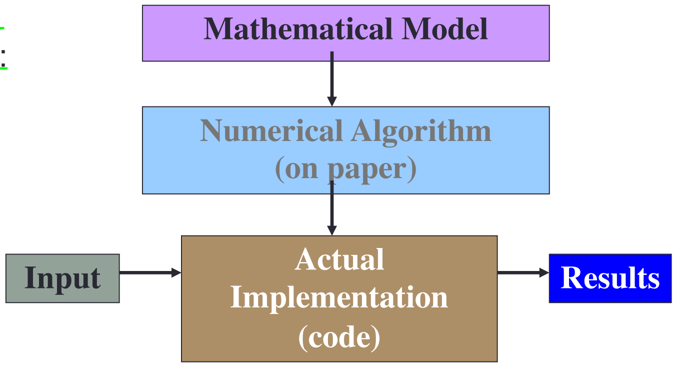
- Half-precision
  - Uses 16-bit
  - Used where high precision is not needed but storage is at a premium
  - Encountered in graphics – e.g. OpenGL, GPUs
- Quadruple (or quad) precision
  - mantissa 112+1 bits, exponent 15 bits
  - ~32 significant figures
- Higher precision?
  - "Multiple precision" libraries exist but SLOW!
  - Some computer algebra packages (e.g. Maple, Mathematica) allow arbitrary precision but these tend to be slow at computation anyway

epcc 26

26

More on errors…

- We want fast, accurate and stable algorithms.
  - But all numerical algorithms produce inaccurate results.
- There are three sources of error:
  - Truncation
  - Rounding
  - Data



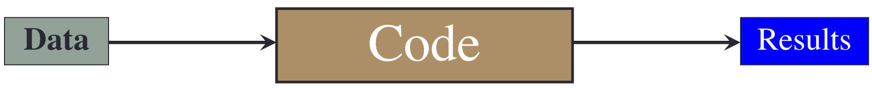
- And there is always a trade-off between speed & accuracy.

epcc 27

27

Data Errors

- Poor data.
  - It is very difficult to write code that will compensate for errors in your input data.
- Badly stored data:
  - The simplest failure can occur on the loading and saving of data.



Inputted and stored as integer or floating-point.

Output from internal representation to decimal (usually).

- Input too fine?
- Output too coarse?

epcc 28

28

Rounding errors: Diagnosis

- How can you tell if your algorithm is suffering due to rounding errors?
  - There is no sure-fire way, but…
  - If you make small changes to the input data, do you see:
    - Wild (chaotic?) variations in the output?
    - No change in the output at all?
  - Do you get the same answer if you change the level of precision? E.g. 32-bit to 64-bit.
  - Are you failing to catch important FP exceptions?
  - Does changing the rounding-mode give you very different answers?

epcc 29

29

Solutions

- Find a better algorithm!
  - Calculating the area of a triangle – e.g. recent practical!
  - However, this could be impossible in a research situation.
- Use a greater level of precision.
  - Up to 128-bit numbers supported on most platforms (slow).
  - Arbitrary precision calculations may also be an option.
  - Beware! The rounding error in not always proportional to the numerical precision!
- Identify the **range** of inputs for which your algorithm **is** reliable.
  - And stick to it!

epcc 30

30

Catastrophic cancellation

- Even subtraction can be dangerous…
  - When the two numbers are nearly equal:
  - E.g. 10000000.23 - 10000000.22 = 0.01
- If both are known precisely, then all is well.
  - We can be sure that the value of 0.01 is free from error.
- If the difference between the two numbers arises from rounding errors, the relative error is magnified.
  - E.g. perhaps the above numbers were supposed to be equal and the small difference was due to rounding error.
  - Result should then have been zero
  - Original values were correct to 9 s.f. (tiny relative error)
  - The relative error in the result is now infinite!
    - Relative error = (0.0 - 0.01)/0.0

epcc

31

31

Truncation Errors

- Rounding errors can be overcome by clever design and using sufficiently large resources.
- This is not true of truncation errors:
  - Due to the differences between the mathematical model and the numerical algorithm.
    - Independent of implementation or precision.
    - You are solving a different system to the one you want.
  - Examples:
    - Truncated Taylor-series.
      - The sin function may be calculated as:
      - $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + O(x^9)$

      How does this approximation affect the accuracy of the algorithm?
    - Discretisation of the problem domain…

epcc

32

32

Discretisation Example

- Heat diffusion over a 1D metal bar:
  - Ends held at two different temperatures, T_1 and T_2.

T_1  T_2

  - This must be discretized to solve it numerically:

T_1  T_2

x_1 x_2 x_3 x_4 … …x_{N-1} x_N

  - You are solving the diffusion equation for a discrete system, not the system you originally wanted. Is this OK?
  - The difference between the results for the original system and the numerical one is the truncation error.

epcc

33

33

Summary

- Floating point numbers defined in IEEE 754 standard
  - defines storage format
  - can be single (32-bit) and double (64-bit) precision
  - 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

epcc

34

34