Physics Coding Club
- Floating Point Numbers

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

- Floating point numbers and representation
- Floating point algebra
- IEE754 to the rescue
- Consequences for numerical programming

Simple observation

- Have you ever wondered why

  0.1 + 0.2 = 0.30000000000000004 ???

- Computers use binary floating point
  - Not every number is representable

    0.1 + 0.4 = 0.5

  - Some numbers ARE representable!
  - Including all integers …
- The answer is the closest number that fits
- Answer has a *rounding err*or as a result

Fixed Point Representation

- How to represent non-integer real numbers in a computer?
  - Could represent as integers with the decimal point after a fixed number of digits, e.g.

12.7 → 0012.7000 stored as 00127000

2345 → 2345.0000 stored as 23450000

0.045 → 0000.0450 stored as 00000450

  - Can include negative numbers using signed (one bit for sign) integers or "two's-compliment"
- But what about 1,349,193? or 0.00000074?
  - Limited range with a given number of digits

A Floating Point Representation

- Base-10 “scientific notation”
  - E.g. write every number as a 4-digit *mantissa* and a 2-digit signed *exponent*, i.e. $\pm 0.xxxx \times 10^{\pm xx}$

  e.g. 0.1000×10^{1} or 0.6626×10^{-33}
  - By convention, the mantissa M is restricted to the range $0.1 \leq M < 1$, i.e. leading digit is zero, hence the *significand* is the mantissa without leading digit
- Hence can only represent 4 million distinct numbers

  10^{4} (with mantissa) x100 (with exponent) x2 (with ± sign for mantissa) x2 (with ± sign for exponent) = 4×10^{6}
- Largest number is 0.9999×10^{99} and smallest is 0.1000×10^{-99} (or 0.0001×10^{-99} if we do not mind losing precision in the mantissa)

Fractions

- Computers work in base-2 and not base-10, which has some interesting consequences for fractions
- Binary numbers e.g.
  - $17_{10} = 16 + 1 = 1x2^4 + 0x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 10001_2$
- Binary fractions follow trivially from decimal

  e.g. $0.625_{10} = 1/2+1/8 = 1x\ 2^{-1} + 0x\ 2^{-2} + 1x\ 2^{-3} = 0.101_2$
- But note that whilst any finite binary fraction is also a finite decimal fraction, the reverse is not true,

  e.g. $0.2_{10} = 1/8 + 1/16 + 1/128 + 1/256 + 1/2048 + 1/4096 + \ldots$

  $0.001100110011\ldots\ldots_2$
- Need infinite computer to store 0.2
  - accept finite precision: $0.001100110011_2 \approx 0.1999511_{10}$

Floating Point Algebra

- Now return to our base-10 floating point format
- Consequence of finite representation is a change to the normal rules of algebra, e.g.
- A+B=A does *not* imply B=0

  $0.1000 \text{x} 10^{+01} + 0.4000 \text{x} 10^{-03} = 0.1000 \text{x} 10^{+01}$

- (A+B)+C is *not* equal to A+(B+C)

  $(0.1000 \text{x} 10^{+01} + 0.4000 \text{x} 10^{-03}) + 0.4000 \text{x} 10^{-03}$

  $= 0.1000 \text{x} 10^{+01} + 0.4000 \text{x} 10^{-03} = 0.1000 \text{x} 10^{+01}$

  but

  $0.1000 \text{x} 10^{+01} + (0.4000 \text{x} 10^{-03} + 0.4000 \text{x} 10^{-03})$

  $= 0.1000 \text{x} 10^{+01} + 0.8000 \text{x} 10^{-03} = 0.1001 \text{x} 10^{+01}$

More Algebra …

- $\sqrt{A^2}$ is *not* equal to $|A|$

  $\sqrt{(0.1000 \times 10^{-60})^2} = \sqrt{(0.0100 \times 10^{-99})} = 0$

- A/B is *not* equal to A * (1/B)

  $0.6000 \times 10^{+01} / 0.7000 \times 10^{+01} = 0.8571 \times 10^{+00}$

  $0.6000 \times 10^{+01} \times (1/0.700 \times 10^{+01}) = 0.6000 \times 10^{+01} \times 0.1429 \times 10^{+00}$

  $= 0.8574 \times 10^{+00}$

- Whilst we have illustrated the point with a simple base-10 encoding, the same is true for any finite representation, including that which is actually used inside most computers …

Consequences

- All of the above has severe consequences for numerical codes
- And if different computer hardware has different internal representations, with different storage formats and rules for handling over/underflow, etc. it can have severe consequences for code portability.
- And accuracy of results!

IEEE 754 Standard

- Before 1985 every computer manufacturer chose their own way of representing floating point numbers based on different trade-offs of speed vs. accuracy
  - No scheme can ever be perfect, but IEEE 754 is based on a lot of experience and is now almost universally used. Benefit is portability of floating point environment – should get repeatable results on any system.
- The standard specifies storage formats, precise specifications for results of operations, special values and specifies runtime behaviour on illegal operations.
- Java does not support IEEE 754 – hence "write once, run anywhere" does *not* imply "write once, get the same results everywhere"!

IEEE Storage Format

	F77†	C	Bits‡	Exponent Bits	Mantissa Bits
Single	Real*4	Float	32	8	24
Double	Real*8	Double	64	11	53
Extended double¶	Real*10	Long double	≥80	≥15	≥64

† F90 has a much nicer, portable way of defining precision using `kind` mechanism

‡ x86 CPU has 80-bit registers which can cause problems if only using first 64-bits with a standard double variable …

¶ Cray traditionally has 64-bit “single” and 128-bit “double” types

IEEE Floating Point Range

	Smallest normalised number*	Largest finite number †	Base-10 accuracy
Single	$1.2*10^{-38}$	$3.4*10^{+38}$	6-9 digits
Double	$2.2*10^{-308}$	$1.8*10^{+308}$	15-17 digits
Extended double	$3.4*10^{-4932}$	$1.2*10^{+4932}$	18-21 digits

* Can also have denormalised numbers that are smaller

† Can also have ∞ if overflow largest number

Remember these limits correspond to largest and smallest numbers representable with BINARY NOT DECIMAL digits.

Different Kinds of Zero

- Can represent 0 exactly with this scheme
  - But what about $0.1000 \times 10^{-99} / 0.1000 \times 10^{+10}$?
  - Whilst the real answer is non-zero it is smaller than the smallest representable number and hence is an *underflow*. It must be represented as zero, but can have a sign!
  - What about $0.1000 \times 10^{-99}/2$?
    - Should it be kept as 0.0500×10^{-99} (breaks the $0.1 \leq M < 1$ rule)
    - or should it be set to zero?
  - The former is an example of a *denormalised* number and can be used to give gradual underflow, but with loss of precision. Can also decide to flush denormals to zero.
  - Some processors barf on denormalised numbers, signal an error and a special software routine is invoked to handle the situation with great cost in time. This, together with loss of precision, is why some CPUs prefer to flush them to zero.

Fortran 90+ Support

- Fortran 90 introduced the *kind* mechanism by which an integer parameter can set floating point format, e.g.
  - default single precision:

```
integer, parameter :: sp = kind(1.0)
```

  - default double precision:

```
integer, parameter :: dp = kind(1.0d0)
```

  - Select a format in which there are *at least* 15 digits of precision in the mantissa and the exponent can be *at least* –300 to +300 in a fully portable way:

```
integer, parameter :: mykind =
  selected_real_kind(p=15,r=300)
```

- Can then declare variables and set values using these formats e.g.

```
real(kind=dp) :: x

x = 1.9763_dp
```

Ensures all arithmetic / storage in double precision.

IEEE Operations

- Standard requires that all simple arithmetic operations return the *nearest representable* number to the true result by default
  - Flexibility for CPU designer in how to do this
  - Consequently $a+b=b+a$ and $a*b=b*a$
- Also specifies rules for truncation and rounding using two *guard bits*

Guard Digits (I)

- Consider our simple format with four digit significand and 2-digit exponent:
- Add $0.5281 \times 10^{+02}$ and $0.9650 \times 10^{+04}$
- Equal exponents : $0.0052 \times 10^{+04} + 0.9650 \times 10^{+04}$
- Add mantissas : $0.9702 \times 10^{+04}$
- Adjust exponent : $0.9702 \times 10^{+04}$

- BUT true answer is $0.9703 \times 10^{+04}$ (round to nearest)!

Guard Digits (II)

- Keep memory storage format the same but use two extra digits *within the CPU's floating point unit:*
- Add $0.5281 \times 10^{+02}$ to $0.9650 \times 10^{+04}$
- Equal exponents : $0.0052\mathbf{81} \times 10^{+04}$ + $0.9650\mathbf{00} \times 10^{+04}$
- Add mantissas : $0.9702\mathbf{81} \times 10^{+04}$
- Adjust exponent : $0.9702\mathbf{81} \times 10^{+04}$
- So now the true answer $0.9703 \times 10^{+04}$ is stored.

IEEE Special Values

- Certain bit-patterns are reserved for representing special values in all bit-lengths
  - Infinity, resulting from overflow, is represented with all mantissa bits=0, all exponent bits=1. Treated according to rules of maths, e.g. dividing a non-zero number by infinity will result in zero.
  - NaN (Not a real Number), resulting from 0/0 etc is represented as infinity with non-zero mantissa. It indicates a value that is not mathematically defined. Any operation involving NaN has NaN as the result.
  - Denormalised numbers have all exponent bits=0 and all bits of mantissa are stored. Can be handled in hardware or software.
  - Zero has all bits set to zero but can have sign bit for ±0

IEEE Exceptions and Traps

- IEEE standard enables programmers to detect when special values are produced, so can write more robust code.
  – Manually write trap handling code for each event for each event, such as overflow to infinity, underflow to zero, division by zero, invalid op, inexact op, etc. Routine names non-standard between compilers/platforms – not portable.
  – Fortran2003 includes this as standard.
  – Can be implemented by compiler itself and have large cost if triggered, so best not! Hence usually off by default.
- NB Traps can also be handled by capturing kernel signals.
  – This is straightforward in C, but can also be done in non-portable way in Fortran by linking to system library `signal` function (Unix and CVF under Win32) to catch SIGFPE = SIGnal Floating Point Exception, etc.

Types of Exception

- Invalid operation, e.g.
  - $\infty - \infty$ or $0*\infty$
  - $0/0$ or ∞/∞
  - $\sqrt{x}$ where $x < 0$
- Division by zero
- Overflow
- Underflow
- Inexact – triggered when precision is lost.

What do we want?

- Generally considered useful for program to abort for overflows or NaN but not for underflow
  - In Fortran, overflow/NaN will usually cause a stop
  - In C, program will carry on regardless – beware!
- NB integer overflow causes wraparound and is usually silent – a 4-byte integer can store 2147483647 to -2147483648, so 2147483647+1 = -2147483648 !
- NB Conversion of floating point to integer, when float is greater than largest possible integer, can do almost anything. In Java, it will simply return the largest possible integer.

So What?

- Implications for optimising compilers
  - Mathematically valid code rearrangements can produce numerically different results
  - Good compilers should have flags that enforce strict IEEE compliance – use them!
  - When turning on optimisation, should always benchmark code against un-optimised version – both for timing and for numerical results – see later lectures.
- Implications for writing code and algorithms
  - **<u>Be VERY careful before use single-precision for mathematical code</u>**
  - Always use appropriate units s.t. quantities are of magnitude ~ unity and not 10^{-34} etc.

Summing Numbers $\sum_{n=1}^{N} \frac{1}{n}$

Consider summing this series forwards (1..N) and backwards (N..1) using single-precision arithmetic

N	Forwards	Backwards	Exact
100	5.187378	5.187377	5.187378
1000	7.485478	7.485472	7.485471
10000	9.787613	9.787604	9.787606
100000	12.09085	12.09015	12.09015
1000000	14.35736	14.39265	14.39273
10000000	15.40368	16.68603	16.69531
100000000	15.40368	18.80792	18.99790

Counting forwards is silly as $15+x=15$ for $x \leq 5\text{x}10^{-7}$ i.e. total stops growing after around 2 million terms

The Logistic Map

$$x_{n+1} = 4x_n\left(1 - x_n\right)$$

n	Single	Double	Correct
0	0.5200000	0.5200000	0.5200000
1	0.9984000	0.9984000	0.9984000
2	0.0063896	0.0063898	0.0063898
3	0.0253952	0.0253957	0.0253957
4	0.0990019	0.0990031	0.0990031
10	0.9957932	0.9957663	0.9957663
20	0.2214707	0.4172717	0.4172717
30	0.6300818	0.0775065	0.0775067
40	0.1077115	0.0162020	0.0161219
50	0.0002839	0.9009089	0.9999786
51	0.0011354	0.3570883	0.0000854

NB Even with only 3 FP operations per cycle, double is doomed after 50 cycles!

Quadratic Formula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

- E.g. $30x^2 + 60.01x + 30.01 = 0$ has roots at

$x = -1$ and $x = -3001/3000$

- Single-precision arithmetic and above formula gives no roots at all!

  30.01 is represented as 30.0100002289

  60.01 is represented as 60.0099983215 hence

  b^2=3601.199899… not 3601.2 and $4ac$=3601.2001

- Using double-precision is not always the answer – the following C program gives no roots with K&R C-compiler and repeated roots with ANSI C!

```
void main(){
   float a=30,b=60.01,c=30.01; double d;
   d=b*b-4*a*c;
   printf("%18.15f\n",d);
}
```

Complex Data Types

- CPU only handles real numbers
  - Certain languages (e.g. FORTRAN) allow a complex data type
  - Other languages (e.g. C and Java) implement it via (sometimes controversial) libraries
- Complex addition/subtraction is simple but not multiplication:

  $(a+\mathrm{i}b) * (c+\mathrm{i}d) = (ac-bd) + (bc+ad)\mathrm{i}$

  - What happens if *ac-bd* is less than the maximum representable number but *ac* is not?
  - What precision problems will we have if $ac \approx bd$?
  - Non-trivial problems to consider …

Complex Division

$$\frac{a+\mathrm{i}b}{c+\mathrm{i}d}=\frac{(ac+bd)+\mathrm{i}(bc-ad)}{c^2+d^2}$$

- This definition is almost useless!
  - If the largest representable number is *Nmax* then this formula will erroneously produce zero when dividing by any complex number z with $|z|>\sqrt{Nmax}$
  - Similarly, if *Nmin* is the smallest representable number then this formula will give ∞ with $|z|<\sqrt{Nmin}$
- There are also issues with disregarding the sign of ±0.0 which results in violation of certain identities, such as $\sqrt{(z^*)} = (\sqrt{z})^*$ whenever $\mathrm{Re}(z)<0.0$
  - Kahan argues this is inevitable in Fortran and C/C++ where complex is implemented as binary (x,y) pair and not as x+iy with a proper imaginary class. F95 can fix this.

Hints: Operations to Avoid

- Divides
  - Multiply by 0.5 instead of dividing by 2.0
- Trig – never do `y=sin(x)**2 ; z=cos(x)**2`

  Instead: `y = sin(x)**2; z = 1.d0 - y`
- Square roots - never do `if (sqrt(x) < y)`
  - Use `if (x < y*y)` if sure about signs.
- Powers – never do `y = x**3.0`
  - Will use microcode / library valid for any non integer power.
  - Use `y = x**3` or `y = x*x*x`

Comparison

- Most floating point numbers are imprecise
  - The same number calculated using two different sums is quite likely to be different

```
A=0.1 + 0.2; B=0.15 + 0.15; Is A=B?
```

- Hence should NEVER test for equality of reals!
  - Simple fix is to test if abs(difference) is small
  - Tricky – how small is small enough?

Error Propagation

- Whilst error in a single FP calculation is small, this can accumulate if repeated
- Need to understand how errors propagate through your algorithm
  - Usually worse with + or - than * or /
  - Understand *epsilon*
  - And different implementations can be either stable or unstable
  - Numerical Analysis

Quadratic Equation revisited

- How to solve $ax^2 + bx + c = 0$?
- Std method: $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
- Alt method: $$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}$$
- Problem: either std or alt method will sometimes fail – when either *a* or *c* (or both) are small then have issues with *b-b* with a significant loss in precision

Quadratic Equation solved

- So need to find a method that does NOT involve *b-b* type expression
- Solution: $$q = -\frac{1}{2}\left[b + \mathrm{sgn}\left(b\right)\sqrt{b^2 - 4ac}\right]$$

$$x = \frac{q}{a}\ and\ x = \frac{c}{q}$$

- E.g. `a=c=1E-6,b=100`; analytic `x2=-1E-8`
- dp math with std formula has `x2=-7.1E-9`; this q form has `x2=-1E-8`

Further Reading

- Chapter 4 of "High Performance Computing (2nd edition)", Kevin Dowd & Charles Severance, O' Reilley (1998).
- "What Every Computer Scientist Should Know About Floating-Point Arithmetic", by David Goldberg in ACM Computing Surveys (Mar 91) http://portal.acm.org/citation.cfm?id=103163
- Walter Kahan's homepage (one of the designers of IEEE 754) http://www.cs.berkeley.edu/~wkahan