UNIVERSITY OF
CAMBRIDGE

Floating Point Computation

(slides 1–123) [with minor fixes]

A six-lecture course

Alan Mycroft

Computer Laboratory, Cambridge University

`http://www.cl.cam.ac.uk/teaching/current/FPComp`

Lent 2008–09

A Few Cautionary Tales

The main enemy of this course is the simple phrase

> "the computer calculated it, so it must be right".

We're happy to be wary for integer programs, e.g. having unit tests to check that

> sorting [5,1,3,2] gives [1,2,3,5],

but then we suspend our belief for programs producing real-number values, especially if they implement a "mathematical formula".

Global Warming

Apocryphal story – Dr X has just produced a new climate modelling program.

Interviewer: what does it predict?

Dr X: Oh, the expected 2–4°C rise in average temperatures by 2100.

Interviewer: is your figure robust?

...

Global Warming (2)

Apocryphal story – Dr X has just produced a new climate modelling program.

Interviewer: what does it predict?

Dr X: Oh, the expected 2–4°C rise in average temperatures by 2100.

Interviewer: is your figure robust?

Dr X: Oh yes, indeed it gives results in the same range even if the input data is randomly permuted ...

We laugh, but let's learn from this.

Global Warming (3)

What could cause this sort or error?

- the wrong mathematical model of reality (most subject areas lack models as precise and well-understood as Newtonian gravity)
- a parameterised model with parameters chosen to fit expected results ('over-fitting')
- the model being very sensitive to input or parameter values
- the discretisation of the continuous model for computation
- the build-up or propagation of inaccuracies caused by the finite precision of floating-point numbers
- plain old programming errors

We'll only look at the last four, but don't forget the first two.

Real world examples

Find Kees Vuik's web page "Computer Arithmetic Tragedies" for these and more:

- Patriot missile interceptor fails to intercept due to 0.1 second being the 'recurring decimal' $0.00\,1100\,1100\ldots_2$ in binary (1991)
- Ariane 5 $500M firework display caused by overflow in converting 64-bit floating-point number to 16-bit integer (1996)
- The Sleipner A offshore oil platform sank . . . post-accident investigation traced the error to inaccurate finite element approximation of the linear elastic model of the tricell (using the popular finite element program NASTRAN). The shear stresses were underestimated by 47% . . .

Learn from the mistakes of the past . . .

Overall motto: threat minimisation

- Algorithms involving floating point (`float` and `double` in Java and C, [misleadingly named] `real` in ML and Fortran) pose a significant threat to the programmer or user.
- Learn to distrust your own naïve coding of such algorithms, and, even more so, get to distrust others'.
- Start to think of ways of sanity checking (by human or machine) any floating point value arising from a computation, library or package—unless its documentation suggests an attention to detail at least that discussed here (and even then treat with suspicion).
- Just because the "computer produces a numerical answer" doesn't mean this has any relationship to the 'correct' answer.

Here be dragons!

What's this course about?

- How computers represent and calculate with *'real number'* values.
- What problems occur due to the values only being finite (both range and precision).
- How these problems add up until you get silly answers.
- How you can stop your programs and yourself from looking silly (and some ideas on how to determine whether existing programs have been silly).
- Chaos and ill-conditionedness.
- Knowing when to call in an expert—remember there is 50+ years of knowledge on this and you only get 6 lectures from me.

UNIVERSITY OF
CAMBRIDGE

Introduction/reminding you what you already know

Reprise: signed and unsigned integers

An 8-bit value such as 10001011 can naturally be interpreted as either an signed number ($2^7 + 2^3 + 2^1 + 2^0 = 139$) or as a signed number ($-2^7 + 2^3 + 2^1 + 2^0 = -117$).

This places the decimal (binary!?!) point at the right-hand end. It could also be interpreted as a *fixed-point number* by imagining a decimal point elsewhere (e.g. in the middle to get) 1000.1011; this would have value $2^3 + 2^{-1} + 2^{-3} + 2^{-4} = 8\frac{11}{16} = 8.6875$.

(The above is an unsigned fixed-point value for illustration, normally we use signed fixed-point values.)

Fixed point values and saturating arithmetic

Fixed-point values are often useful (e.g. in low-power/embedded devices) but they are prone to overflow. E.g. 2*10001011 = 00010110 so 2*8.6875 = 1.375!! One alternative is to make operations *saturating* so that 2*10001011 = 11111111 which can be useful (e.g. in audio). Note $1111.1111_2 = 15.9375_{10}$.

An alternative way to avoid this sort of overflow is to allow the decimal point to be determined at run-time (by another part of the value) "floating point" instead of being fixed (independent of the value as above) "fixed point" – the subject of this course.

Back to school

Scientific notation (from Wikipedia, the free encyclopedia)

In *scientific notation*, numbers are written using powers of ten in the form $a \times 10^b$ where b is an integer *exponent* and the *coefficient* a is any real number, called the *significand* or *mantissa*.

In *normalised form*, a is chosen such that $1 \leq a < 10$. It is implicitly assumed that scientific notation should always be normalised except during calculations or when an unnormalised form is desired.

What Wikipedia should say: zero is problematic—its exponent doesn't matter and it can't be put in normalised form.

Back to school (2)

Multiplication and division (from Wikipedia, with some changes)

Given two numbers in scientific notation,

$$x_0 = a_0 \times 10^{b_0} \qquad x_1 = a_1 \times 10^{b_1}$$

Multiplication and division;

$$x_0 * x_1 = (a_0 * a_1) \times 10^{b_0+b_1} \qquad x_0/x_1 = (a_0/a_1) \times 10^{b_0-b_1}$$

Note that result is not guaranteed to be normalised even if inputs are: $a_0 * a_1$ may now be between 1 and 100, and a_0/a_1 may be between 0.1 and 10 (both at most one out!). E.g.

$$\begin{aligned} 5.67 \times 10^{-5} * 2.34 \times 10^{2} &\approx 13.3 \times 10^{-3} = 1.33 \times 10^{-2} \\ 2.34 \times 10^{2} / 5.67 \times 10^{-5} &\approx 0.413 \times 10^{7} = 4.13 \times 10^{6} \end{aligned}$$

Back to school (3)

Addition and subtraction require the numbers to be represented using the same exponent, normally the bigger of b_0 and b_1.
W.l.o.g. $b_0 > b_1$, so write $x_1 = (a_1 * 10^{b_1 - b_0}) \times b_0$ (a shift!) and add/subtract the mantissas.

$$x_0 \pm x_1 = (a_0 \pm (a_1 * 10^{b_1 - b_0})) \times 10^{b_0}$$

E.g.

$$2.34 \times 10^{-5} + 5.67 \times 10^{-6} = 2.34 \times 10^{-5} + 0.567 \times 10^{-5} \approx 2.91 \times 10^{-5}$$

A cancellation problem we will see more of:

$$2.34 \times 10^{-5} - 2.33 \times 10^{-5} = 0.01 \times 10^{-5} = 1.00 \times 10^{-7}$$

When numbers reinforce (e.g. add with same-sign inputs) new mantissa is in range $[1, 20)$, when they cancel it is in range $[0..10)$. After cancellation we may require several shifts to normalise.

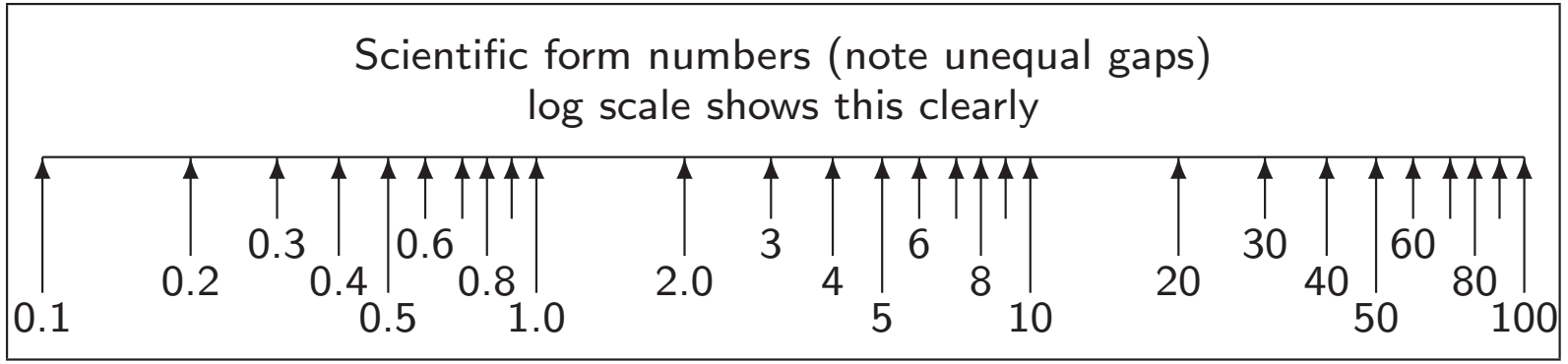
Significant figures can mislead

When using scientific-form we often compute repeatedly keeping the same number of digits in the mantissa. In science this is often the number of digits of accuracy in the original inputs—hence the term (decimal) *significant figures* (sig.figs. or sf).

This is risky for two reasons:

- As in the last example, there may be 3sf in the result of a computation but little *accuracy* left.
- 1.01×10^1 and 9.98×10^0 are quite close, and both have 3sf, but changing them by one *ulp* ('unit in last place') changes the value by nearly 1% (1 part in 101) in the former and about 0.1% (1 part in 998) in the latter. Later we'll prefer "relative error".

Significant figures can mislead (2)



You might prefer to say *sig.figs.*$(4.56) = -\log_{10} 0.01/4.56$ so that $sf\,(1.01)$ and $sf\,(101)$ is about 3, and $sf\,(9.98)$ and $sf\,(0.0000998)$ is nearly 4. (BTW, a good case can be made for 2 and 3 respectively instead.)

Exercise: with this more precise understanding of sig.figs. how do the elementary operations $(+, -, *, /$; operating on nominal 3sf arguments to give a nominal 3sf result) really behave?

Get your calculator out!

Calculators are just floating point computers. Note that physical calculators often work in decimal, but calculator programs (e.g. `xcalc`) often work in binary. Many interesting examples on this course can be demonstrated on a calculator—the underlying problem is floating point computation and naïve-programmer failure to understand it rather than programming *per se*.

Amusing to try (computer output is red)

$$(1 + \texttt{1e20}) - \texttt{1e20} = \texttt{0.000000} \qquad 1 + (\texttt{1e20} - \texttt{1e20}) = \texttt{1.000000}$$

But *everyone knows* that $(a + b) + c = a + (b + c)$ (associativity) in maths and hence $(a + b) - d = a + (b - d)$ [just write $d = -c$]!!!

Get your calculator out (2)

How many sig.figs. does it work to/display [example is `xcalc`]?

```
1 / 9 =                  0.11111111
<ans> - 0.11111111 =     1.111111e-09
<ans> - 1.111111e-9 =    1.059003e-16
<ans> - 1.059e-16 =      3.420001e-22
```

Seems to indicate 16sf calculated (16 ones before junk appears) and 7/8sf displayed—note the endearing/buggy(?) habit of `xcalc` of displaying one fewer sf when an exponent is displayed.

Stress test it:

$$\texttt{sin 1e40} = 0.3415751$$

Does anyone believe this result? [Try your calculator/programs on it.]

Computer Representation

A computer representation must be finite. *If* we allocate a fixed size of storage for each then we need to

- fix a size of mantissa (sig.figs.)
- fix a size for exponent (exponent range)

We also need to agree what the number means, e.g. agreeing the base used by the exponent.

Why "floating point"? Because the exponent logically determines where the decimal point is placed within (or even outside) the mantissa. This originates as an opposite of "fixed point" where a 32-bit integer might be treated as having a decimal point between (say) bits 15 and 16.

Floating point can simple be thought of simply as (a subset of all possible) values in scientific notation held in a computer.

Computer Representation (2)

Nomenclature. Given a number represented as $\beta^e \times d_0.d_1 \cdots d_{p-1}$ we call β the base (or radix) and p the precision.

Note that $(\beta = 2, p = 24)$ may be less accurate than $(\beta = 10, p = 10)$.

Now let's turn to binary representations ($\beta = 2$, as used on most modern machines, will solely be considered in this course, but note the IBM/360 series of mainframes used $\beta = 16$ which gave some entertaining but obsolete problems).

Decimal or Binary?

Most computer arithmetic is in binary, as values are represented as 0's and 1's. However, floating point values, even though they are represented as bits, can use $\beta = 2$ or $\beta = 10$.

Most computer programs use the former. However, to some extent it is non-intuitive for humans, especially for fractional numbers (e.g. that 0.1_{10} is recurring when expressed in binary).

Most calculators work in decimal $\beta = 10$ to make life more intuitive, but various software calculators work in binary (see `xcalc` above).

Microsoft Excel is a particular issue. It tries hard to give the illusion of having floating point with 15 decimal digits, but internally it uses 64-bit floating point. This gives rise to a fair bit of criticism and various users with jam on their faces.

UNIVERSITY OF
CAMBRIDGE

Floating point representation

Standards

In the past every manufacturer produced their own floating point hardware and floating point programs gave different answers. IEEE standardisation fixed this.

There are two different IEEE standards for floating-point computation.

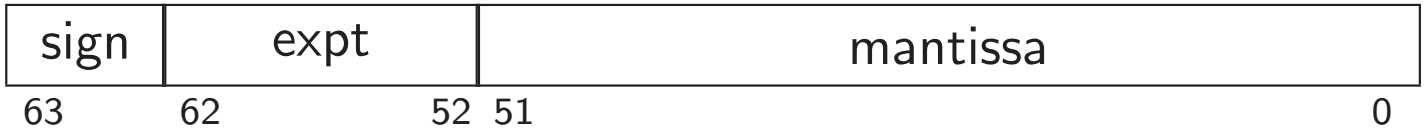
IEEE 754 is a binary standard that requires $\beta = 2, p = 24$ (number of mantissa bits) for *single precision* and $p = 53$ for *double precision*. It also specifies the precise layout of bits in a single and double precision. [Edited quote from Goldberg.]
It has just (2008) been revised to include additional (longer) binary floating point formats and also decimal floating formats.

IEEE 854 is more general and allows binary and decimal representation without fixing the bit-level format.

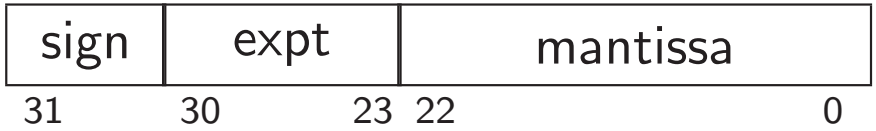
IEEE 754 Floating Point Representation

Actually, I'm giving the version used on x86 (similar issues arise as in the ordering of bytes with an 32-bit integer).

Double precision: 64 bits (1+11+52), $\beta = 2, p = 53$



Single precision: 32 bits (1+8+23), $\beta = 2, p = 24$



Value represented is *typically*: $(s\ ?\ -1 : 1) * 1.mmmmmm * 2^{eeeee}$.

Note *hidden bit*: 24 (or 53) sig.bits, only 23 (or 52) stored!

Hidden bit and exponent representation

Advantage of base-2 ($\beta = 2$) exponent representation: all normalised numbers start with a '1', so no need to store it. (Just like base 10, there normalised numbers start 1..9, in base 2 they start 1..1.)

Hidden bit and exponent representation (2)

But: what about the number zero? Need to cheat, and while we're at it we create representations for infinity too. In single precision:

exponent (binary)	exponent (decimal)	value represented
00000000	0	zero if $mmmmm = 0$ ('denormalised number' otherwise)
00000001	1	$1.mmmmmm * 2^{-126}$
...	...	...
01111111	127	$1.mmmmmm * 2^{-0} = 1.mmmmmm$
10000000	128	$1.mmmmmm * 2^{1}$
...	...	...
11111110	254	$1.mmmmmm * 2^{127}$
11111111	255	infinity if $mmmmm = 0$ ('NaN's otherwise)

Digression (non-examinable)

IEEE define terms e_{min}, e_{max} delimiting the exponent range and programming languages define constants like

```
#define FLT_MIN_EXP (-125)
#define FLT_MAX_EXP 128
```

whereas on the previous slide I listed the min/max exponent uses as $1.mmmmmm * 2^{-126}$ to $1.mmmmmm * 2^{127}$.

BEWARE: IEEE and ISO C write the above ranges as $0.1mmmmmm * 2^{-125}$ to $0.1mmmmmm * 2^{128}$ (so all p digits are *after* the decimal point) so all is consistent, but remember this if you ever want to use `FLT_MIN_EXP` or `FLT_MAX_EXP`.

I've kept to the more intuitive $1.mmmmm$ form in these notes.

Hidden bit and exponent representation (3)

Double precision is similar, except that the 11-bit exponent field now gives non-zero/non-infinity exponents ranging from 000 0000 0001 representing 2^{-1022} via 011 1111 1111 representing 2^0 to 111 1111 1110 representing 2^{1023}.

This representation is called "excess-127" (single) or "excess-1023" (double precision).

Why use it?

Because it means that (for positive numbers, and ignoring NaNs) floating point comparison is the same as integer comparison.

Why 127 not 128? The committee decided it gave a more symmetric number range (see next slide).

Solved exercises

What's the smallest and biggest normalised numbers in single precision IEEE floating point?

Biggest: exponent field is 0..255, with 254 representing 2^{127}. The biggest mantissa is 1.111...111 (24 bits in total, including the implicit leading zero) so $1.111...111 \times 2^{127}$. Hence almost 2^{128} which is $2^8 * 2^{120}$ or $256 * 1024^{12}$, i.e. around $3 * 10^{38}$.

`FLT_MAX` from `<float.h>` gives `3.40282347e+38f`.

Smallest? That's easy: $-3.40282347\text{e}{+38}$! OK, I meant smallest positive. I get $1.000...000 \times 2^{-126}$ which is by similar reasoning around 16×2^{-130} or 1.6×10^{-38}.

`FLT_MIN` from `<float.h>` gives `1.17549435e-38f`.

Solved exercises (2)

[Not part of this course: *'denormalised numbers'* can range down to $2^{-150} \approx 1.401298$e-45, but there is little accuracy at this level.]

And the precision of single precision? 2^{23} is about 10^7, so in principle 7sf. (But remember this is for representing a single number, operations will rapidly chew away at this.)

And double precision? `DBL_MAX 1.7976931348623157e+308` and `DBL_MIN 2.2250738585072014e-308` with around 16sf.

How many single precision floating point numbers are there?
Answer: 2 signs * 254 exponents * 2^{23} mantissas for normalised numbers plus 2 zeros plus 2 infinities (plus NaNs and denorms not covered in this course).

Solved exercises (3)

Which values are representable *exactly* as normalised single precision floating point numbers?

Legal Answer: $\pm\left((2^{23}+i)/2^{23}\right)\times 2^{j}$ where $0\leq i<2^{23}$ and $-126\leq j\leq 127$ (because of hidden bit)
(corrected from first version of these notes)

More Useful Answer: $\pm\, i\times 2^{j}$ where $0\leq i<2^{24}$ and $-126-23\leq j\leq 127-23$
(but legally only right for $j<-126$ if we also include denormalised numbers):

Compare: what values are exactly representable in normalised 3sf decimal? Answer: $i\times 10^{j}$ where $100\leq i<1000$.

Solved exercises (4)

UNIVERSITY OF CAMBRIDGE

So you mean 0.1 is not exactly representable? Its nearest single precision IEEE number is the *recurring* 'decimal' 0x3dcccccd (note round to nearest) i.e. $\boxed{0}\boxed{011\ 1101\ 1}\boxed{100\ 1100\ 1100\ 1100\ 1100\ 1101}$. Decoded, this is $2^{-4} \times 1.100\,1100 \cdots 1101_2$, or $\frac{1}{16} \times (2^0 + 2^{-1} + 2^{-4} + 2^{-5} + \cdots + 2^{-23})$. It's a geometric progression (that's what 'recurring decimal' means) so you can sum it exactly, but just from the first two terms it's clear it's approximately $1.5/16$.

This is the source of the Patriot missile bug.

Solved exercises (5)

BTW, So how many times does

```
for (f = 0.0; f < 1.0; f += 0.1) { C }
```

iterate? Might it differ if `f` is single/double?

NEVER count using floating point unless you really know what you're doing (and write a comment half-a-page long explaining to the 'maintenance programmer' following you why this code works and why naïve changes are likely to be risky).

See `sixthcounting.c` for a demo where `float` gets this accidentally right and `double` gets it accidentally wrong.

Apology: many of the examples are still written in C from last year – I'll write Java versions of them (prompt me if necessary!).

Solved exercises (5)

How many sig.figs. do I have to print out a single-precision float to be able to read it in again exactly?

Answer: The smallest (relative) gap is from 1.111110 to 1.1111111, a difference of about 1 part in 2^{24}. If this of the form $1.xxx \times 10^b$ when printed in decimal then we need 9 sig.figs. (including the leading '1', i.e. 8 after the decimal point in scientific notation) as an ulp change is 1 part in 10^8 and $10^7 \leq 2^{24} \leq 10^8$.

[But you may only need 8 sig.figs if the decimal starts with 9.xxx—see `printsigfig_float.c`].

Note that this is significantly more than the 7sf accuracy quoted earlier for float!

Signed zeros, signed infinities

Signed zeros can make sense: if I repeatedly divide a positive number by two until I get zero (*'underflow'*) I might want to remember that it started positive, similarly if I repeatedly double a number until I get *overflow* then I want a signed infinity.

However, while differently-signed zeros compare equal, not all 'obvious' mathematical rules remain true:

```
int main() {
  double a = 0, b = -a;
  double ra = 1/a, rb = 1/b;
  if (a == b && ra != rb)
     printf("Ho hum a=%f == b=%f but 1/a=%f != 1/b=%f\n", a,b, ra,rb);
  return 0; }
```

Gives:

```
Ho hum a=0.000000 == b=-0.000000 but 1/a=inf != 1/b=-inf
```

Why infinities and NaNs?

The alternatives are to give either a wrong value, or an exception.

An infinity (or a NaN) propagates 'rationally' through a calculation and enables (e.g.) a matrix to show that it had a problem in calculating some elements, but that other elements can still be OK.

Raising an exception is likely to abort the whole matrix computation and giving wrong values is just plain dangerous.

The most common way to get a NaN is by calculating 0.0/0.0 (there's no obvious 'better' interpretation of this) and library calls like `sqrt(-1)` generally also return NaNs (but tools using scripting languages can return $0 + 1i$ if, unlike Java, they are untyped and so don't need the result to fit in a single floating point variable).

IEEE 754 History

Before IEEE 754 almost every computer had its own floating point format with its own form of rounding – so floating point results differed from machine to machine!

The IEEE standard largely solved this (in spite of mumblings "this is too complex for hardware and is too slow" – now obviously proved false). In spite of complaints (e.g. the two signed zeros which compare equal but which can compare unequal after a sequence of operators) it has stood the test of time.

However, many programming language standards allow intermediate results in expressions to be calculated at higher precision than the programmer requested so `f(a*b+c)` and `{ float t=a*b; f(t+c); }` may call `f` with different values. (Sigh!)

IEEE 754 and Intel x86

Intel had the first implementation of IEEE 754 in its 8087 co-processor chip to the 8086 (and drove quite a bit of the standardisation). However, while this x87 chip *could* implement the IEEE standard compiler writers and others used its internal 80-bit format in ways forbidden by IEEE 754 (preferring speed over accuracy!).

The SSE2 instruction set on modern Intel x86 architectures (Pentium and Core 2) includes a separate (better) instruction set which better enables compiler writers to generate fast (and IEEE-valid) floating-point code.

BEWARE: most modern x86 computers therefore have *two* floating point units! This means that a given program for a Pentium (depending on whether it is compiled for the SSE2 or x87 instruction set) can produce different answers; the default often depends on whether the host computer is running in 32-bit mode or 64-bit mode. (Sigh!)

UNIVERSITY OF
CAMBRIDGE

Floating point operations

IEEE arithmetic

This is a very important slide.

IEEE basic operations ($+, -, *, /$ are defined as follows):
Treat the operands (IEEE values) as precise, do perfect mathematical operations on them (NB the result might not be representable as an IEEE number, analogous to 7.47+7.48 in 3sf decimal). Round(*) this mathematical value to the *nearest* representable IEEE number and store this as result. In the event of a tie (e.g. the above decimal example) chose the value with an even (i.e. zero) least significant bit.

[This last rule is statistically fairer than the "round down 0–4, round up 5–9" which you learned in school. *Don't be tempted to believe the exactly 0.50000 case is rare!*]

This is a very important slide.

[(*) See next slide]

IEEE Rounding

In addition to rounding prescribed above (which is the default behaviour) IEEE requires there to be a global flag which can be set to one of 4 values:

Unbiased which rounds to the nearest value, if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit. This mode is required to be default.

Towards zero

Towards positive infinity

Towards negative infinity

Be very sure you know what you are doing if you change the mode, or if you are editing someone else's code which exploits a non-default mode setting.

Other mathematical operators?

Other mathematical operators are typically implemented in libraries. Examples are `sin`, `sqrt`, `log` etc. It's important to ask whether implementations of these satisfy the IEEE requirements: e.g. does the `sin` function give the nearest floating point number to the corresponding perfect mathematical operation's result when acting on the floating point operand treated as perfect? [This would be a perfect-quality library with error within 0.5 ulp and still a research problem for most functions.]

Or is some lesser quality offered? In this case a library (or package) is only as good as the vendor's careful explanation of what error bound the result is accurate to (remember to ask!). ± 1 ulp is excellent.

But remember (see 'ill-conditionedness' later) that a more important practical issue might be how a change of 1 ulp on the input(s) affects the output – and hence how input error bars become output error bars.

The java.lang.Math libraries

Java (http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Math.html) has quite well-specified math routines, e.g. for `asin()` "arc sine"

"Returns the arc sine of an angle, in the range of -pi/2 through pi/2. Special cases:

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

A result must be within 1 ulp of the correctly rounded result.
[perfect accuracy requires result within 0.5 ulp]

Results must be semi-monotonic.
[i.e. given that arc sine is monotonically increasing, this condition requires that $x < y$ implies $\texttt{asin}(x) \leq \texttt{asin}(y)$]"

Errors in Floating Point

When we do a floating point computation, errors (w.r.t. perfect mathematical computation) essentially arise from two sources:

- the inexact representation of constants in the program and numbers read in as data. (Remember even 0.1 in decimal cannot be represented exactly in as an IEEE value, just like 1/3 cannot be represented exactly as a finite decimal. Exercise: write 0.1 as a (recurring) binary number)
- rounding errors produced by (in principle) every IEEE operation.

These errors build up during a computation, and we wish to be able to get a bound on them (so that we know how accurate our computation is).

Errors in Floating Point (2)

It is useful to identify two ways of measuring errors. Given some value a and an approximation b of a, the

Absolute error is $\epsilon = |a - b|$

Relative error is $\eta = \dfrac{|a - b|}{|a|}$

[`http://en.wikipedia.org/wiki/Approximation_error`]

Errors in Floating Point (3)

Of course, we don't normally know the *exact* error in a program, because if we did then we could calculate the floating point answer and add on this known error to get a mathematically perfect answer!

So, when we say the "relative error is (say) 10^{-6}" we mean that the true answer lies within the range $[(1 - 10^{-6})v..(1 + 10^{-6})v]$

$x \pm \epsilon$ is often used to represent any value in the range $[x - \epsilon..x + \epsilon]$.

This is the idea of "error bars" from the sciences.

Errors in Floating Point Operations

Errors from $+, -$: these sum the absolute errors of their inputs

$$(x \pm \epsilon_x) + (y \pm \epsilon_y) = (x + y) \pm (\epsilon_x + \epsilon_y)$$

Errors from $*, /$: these sum the relative errors (if these are small)

$$(x(1 \pm \eta_x)) * (y(1 \pm \eta_y)) = (x * y)(1 \pm (\eta_x + \eta_y) \pm \eta_x \eta_y)$$

and we discount the $\eta_x \eta_y$ product as being negligible.

If the justifications trouble you, then ask your supervisor—you don't need to be able to reproduce them for this course.

Beware: when addition or subtraction causes partial or total cancellation the relative error of the result can be much larger than that of the operands.

Gradual loss of significance

Consider the program (see the calculator example earlier)

```
double x = 10.0/9.0;
for (i=0; i<30; i++)
{   printf("%e\n", x);
    x = (x-1.0) * 10.0;
}
```

Initially x has around 16sf of accuracy (IEEE `double`). But after every cycle round the loop it still stores 16sf, but the accuracy of the stored value reduces by 1sf per iteration. [Try it!]

This is called "gradual loss of significance" and is in practice at least as much a problem as overflow and underflow and *much* harder to identify.

Output

1.111111e+00	1.111112e+00	4.935436e+03
1.111111e+00	1.111116e+00	4.934436e+04
1.111111e+00	1.111160e+00	4.934336e+05
1.111111e+00	1.111605e+00	4.934326e+06
1.111111e+00	1.116045e+00	4.934325e+07
1.111111e+00	1.160454e+00	4.934325e+08
1.111111e+00	1.604544e+00 [≈16sf]	4.934325e+09
1.111111e+00	6.045436e+00	4.934325e+10
1.111111e+00	5.045436e+01	4.934325e+11
1.111111e+00	4.945436e+02	4.934325e+12

Note that *maths* says every number is in the range [1,10)!

Machine Epsilon

Machine epsilon is defined as the difference between 1.0 and the smallest *representable* number which is greater than one, i.e. 2^{-23} in single precision, and 2^{-52} in double (in both cases $\beta^{-(p-1)}$). ISO 9899 C says:

> "the difference between 1 and the least value greater than 1 that is representable in the given floating point type"

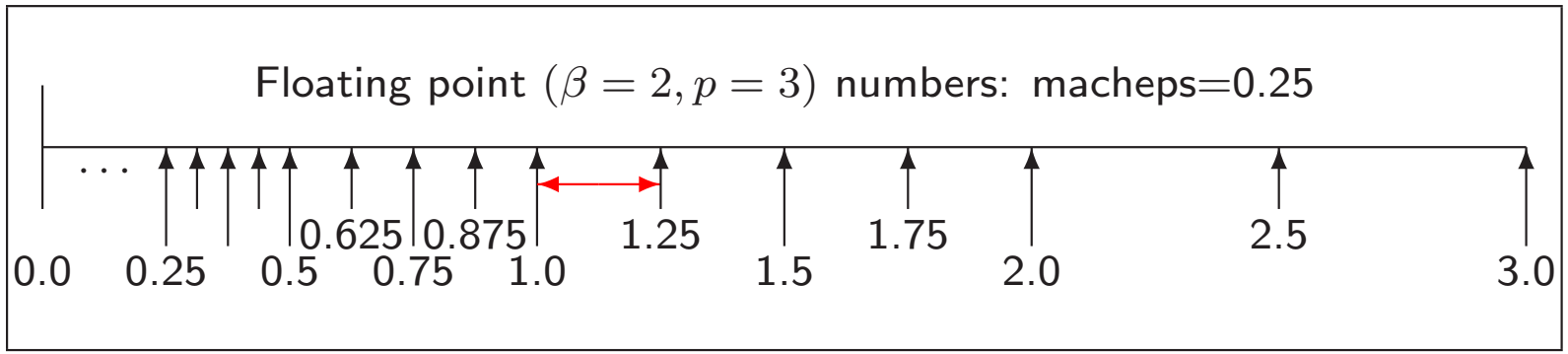
I.e. machine epsilon is 1 ulp for the representation of 1.0.

For IEEE arithmetic, the C library `<float.h>` defines

```
#define FLT_EPSILON         1.19209290e-7F
#define DBL_EPSILON         2.220446049250313e-16
```

Machine Epsilon (2)

Machine epsilon is useful as it gives an upper bound on the relative error caused by getting a floating point number wrong by 1 ulp, and is therefore useful for expressing errors independent of floating point size. (The relative error caused by being wrong by 1 ulp *can* be up to 50% smaller than this, consider 1.5 or 1.9999.)



Machine Epsilon (3)

Some sources give an alternative (bad) definition: "the smallest number which when added to one gives a number greater than one". (With rounding-to-nearest this only needs to be slightly more than half of our machine epsilon).

Microsoft MSDN documentation (Feb 2009) gets this wrong:

Constant	Value	Meaning
FLT_EPSILON	1.192092896e-07F	Smallest such that 1.0+FLT_EPSILON !=1.0

The value is right by the C standard, but the explanation inconsistent with it – due to rounding. Whoops:

```
  float one = 1.0f, xeps = 0.7e-7f;
  printf("%.7e + %.7e = %.7e\n", one, xeps, (float)(xeps+one));
===>>>>  1.0000000e+00 + 6.9999999e-08 = 1.0000001e+00
```

Machine Epsilon (4)

Oh yes, the GNU C library documentation (Nov 2007) gets it wrong too:

```
FLT_EPSILON
    This is the minimum positive floating point number of type
    float such that 1.0 + FLT_EPSILON != 1.0 is true.
```

Again, the implemented value is right, but the explanation inconsistent with it.

Is the alternative definition 'bad'? It's sort-of justifiable as almost the maximum quantisation error in 'round-to-nearest', but the ISO standards chose to use "step between adjacent values" instead.

Machine Epsilon (5) – Negative Epsilon

We defined machine epsilon as the difference between 1.0 and the smallest *representable* number which is greater than one.

What about the difference between 1.0 and the greatest *representable* number which is smaller than one?

In IEEE arithmetic this is exactly 50% of machine epsilon.

Why? Witching-hour effect. Let's illustrate with precision of 5 binary places (4 stored). One is $2^0 \times 1.0000$, the next smallest number is $2^0 \times 0.111111111\ldots$ truncated to fit. But when we write this normalised it is $2^{-1} \times 1.1111$ and so its ulp represents only half as much as the ulp in $2^0 \times 1.0001$.

Revisiting `sin 1e40`

The answer given by `xcalc` earlier is totally bogus. Why?

10^{40} is stored (like all numbers) with a relative error of around machine epsilon. (So changing the stored value by 1 ulp results in an absolute error of around $10^{40} \times machine\_epsilon$.) Even for `double` (16sf), this absolute error of representation is around 10^{24}. But the `sin` function cycles every 2π. So we can't even represent which of many billions of cycles of sine that 10^{40} should be in, let alone whether it has any sig.figs.!

On a decimal calculator 10^{40} is stored accurately, but I would need π to 50sf to have 10sf left when I have range-reduced 10^{40} into the range $[0, \pi/2]$. So, who *can* calculate `sin 1e40`? Volunteers?

UNIVERSITY OF
CAMBRIDGE

Simple maths, simple programs

Non-iterative programs

Iterative programs need additional techniques, because the program may be locally sensible, but a small representation or rounding error can slowly grow over many iterations so as to render the result useless.

So let's first consider a program with a fixed number of operations:

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

Or in C/Java:

```
double root1(double a, double b, double c)
{   return (-b + sqrt(b*b - 4*a*c))/(2*a);   }
double root2(double a, double b, double c)
{   return (-b - sqrt(b*b - 4*a*c))/(2*a);   }
```

What could be wrong with this so-simple code?

Solving a quadratic

Let's try to make sure the *relative error* is small.

Most operations in $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ are multiply, divide, sqrt—these add little to relative error (see earlier): unary negation is harmless too.

But there are two additions/subtractions: $b^2 - 4ac$ and $-b \pm \sqrt{(\cdots)}$.

Cancellation in the former ($b^2 \approx 4ac$) is not too troublesome (why?), but consider what happens if $b^2 \gg 4ac$. This causes the latter $\pm$ to be problematic for one of the two roots.

Just consider $b > 0$ for now, then the problem root (the smaller one in magnitude) is $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$.

Getting the small root right

$$\begin{aligned} x &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\ &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} \\ &= \frac{-2c}{b + \sqrt{b^2 - 4ac}} \end{aligned}$$

These are all equal in *maths*, but the final expression *computes* the little root much more accurately (no cancellation if $b > 0$).

But keep the big root *calculation* as

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

Need to do a bit more (i.e. opposite) work if $b < 0$

Illustration—quadrat_float.c


```
1 x^2 + -3 x + 2 =>
root1 2.000000e+00 (or 2.000000e+00 double)

1 x^2 + 10 x + 1 =>
root1 -1.010203e-01 (or -1.010205e-01 double)

1 x^2 + 100 x + 1 =>
root1 -1.000214e-02 (or -1.000100e-02 double)

1 x^2 + 1000 x + 1 =>
root1 -1.007080e-03 (or -1.000001e-03 double)

1 x^2 + 10000 x + 1 =>
root1 0.000000e+00 (or -1.000000e-04 double)
root2 -1.000000e+04 (or -1.000000e+04 double)
```

Summing a Finite Series

We've already seen $(a + b) + c \neq a + (b + c)$ in general. So what's the best way to do (say)

$$\sum_{i=0}^{n} \frac{1}{i + \pi} \; ?$$

Of course, while this formula mathematically sums to infinity for $n = \infty$, if we calculate, using `float`

$$\frac{1}{0 + \pi} + \frac{1}{1 + \pi} + \cdots + \frac{1}{i + \pi} + \cdots$$

until the sum stops growing we get 13.8492260 after 2097150 terms.

But is this correct? Is it the only answer?

[See sumfwdback.c on the course website for a program.]

Summing a Finite Series (2)

Previous slide said: using `float` with $N = 2097150$ we got

$$\frac{1}{0+\pi} + \frac{1}{1+\pi} + \cdots + \frac{1}{N+\pi} = 13.8492260$$

But, by contrast

$$\frac{1}{N+\pi} + \cdots + \frac{1}{1+\pi} + \frac{1}{0+\pi} = 13.5784464$$

Using double precision (64-bit floating point) we get:

- forward: 13.5788777897524611
- backward: 13.5788777897519921

So the backwards one *seems* better. But why?

Summing a Finite Series (3)

When adding $a + b + c$ it is generally more accurate to sum the smaller two values and then add the third. Compare $(14 + 14) + 250 = 280$ with $(250 + 14) + 14 = 270$ when using 2sf decimal – and carefully note where rounding happens.

So summing backwards is best for the (decreasing) series in the previous slide.

Summing a Finite Series (4)

General tips for an accurate result:

- Sum starting from smallest
- Even better, take the two smallest elements and replace them with their sum (repeat until just one left).
- If some numbers are negative, add them with similar-magnitude positive ones first (this reduces their magnitude without losing accuracy).

Summing a Finite Series (5)

A neat general algorithm (non-examinable) is *Kahan's summation algorithm*:

`http://en.wikipedia.org/wiki/Kahan_summation_algorithm`

This uses a second variable which approximates the error in the previous step which can then be used to compensate in the next step.

For the example in the notes, even using single precision it gives:

- Kahan forward: 13.5788774
- Kahan backward: 13.5788774

which is within one ulp of the double sum 13.57887778975...

A chatty article on summing can be found in Dr Dobb's Journal:
`http://www.ddj.com/cpp/184403224`

UNIVERSITY OF
CAMBRIDGE

Infinitary/limiting computations

Rounding versus Truncation Error

Many mathematical processes are infinitary, e.g. limit-taking (including differentiation, integration, infinite series), and iteration towards a solution.

There are now two logically distinct forms of error in our calculations

Rounding error the error we get by using finite arithmetic during a computation. [We've talked about this exclusively until now.]

Truncation error (a.k.a. discretisation error) the error we get by stopping an infinitary process after a finite point. [This is new]

Note the general antagonism: the finer the mathematical approximation the more operations which need to be done, and hence the worse the accumulated error. Need to compromise, or *really* clever algorithms (beyond this course).

Illustration—differentiation

Suppose we have a nice civilised function f (we're not even going to look at malicious ones). By civilised I mean smooth (derivatives exist) and $f(x)$, $f'(x)$ and $f''(x)$ are around 1 (i.e. between, say, 0.1 and 10 rather than 10^{15} or 10^{-15} or, even worse, 0.0). Let's suppose we want to *calculate* an approximation to $f'(x)$ at $x = 1.0$ given only the code for f.

Mathematically, we define

$$f'(x) = \lim_{h \to 0} \left(\frac{f(x+h) - f(x)}{h} \right)$$

So, we just calculate $(f(x+h) - f(x))/h$, don't we?

Well, just how do we choose h? Does it matter?

Illustration—differentiation (2)

The maths for $f'(x)$ says take the limit as h tends to zero. But if h is smaller than *machine epsilon* (2^{-23} for `float` and 2^{-52} for `double`) then, for x about 1, $x + h$ will compute to the same value as x. So $f(x + h) - f(x)$ will evaluate to zero!

There's a more subtle point too, if h is small then $f(x + h) - f(x)$ will produce lots of cancelling (e.g. $1.259 - 1.257$) hence a high relative error (few sig.figs. in the result).
'Rounding error.'

But if h is too big, we also lose: e.g. dx^2/dx at 1 should be 2, but taking $h = 1$ we get $(2^2 - 1^2)/1 = 3.0$. Again a high relative error (few sig.figs. in the result).
'Truncation error.'

Illustration—differentiation (3)

Answer: the two errors vary oppositely w.r.t. h, so compromise by making the two errors of the same order to minimise their total effect.

The truncation error can be calculated by Taylor:

$$f(x+h) = f(x) + hf'(x) + h^2 f''(x)/2 + O(h^3)$$

So the truncation error in the formula is approximately $hf''(x)/2$ (check it yourself), i.e. about h given the assumption on f'' being around 1.

Illustration—differentiation (4)

For rounding error use Taylor again, and allow a minimal error of $macheps$ to creep into f and get (remember we're also assuming $f(x)$ and $f'(x)$ is around 1, and we'll write $macheps$ for $machine\_epsilon$):

$$\begin{aligned}(f(x+h)-f(x))/h &= (f(x)+hf'(x)\pm macheps - f(x))/h \\ &= 1 \pm macheps/h\end{aligned}$$

So the *rounding error* is $macheps/h$.

Equating rounding and truncation errors gives $h = macheps/h$, i.e. $h = \sqrt{macheps}$ (around 3.10^{-4} for single precision and 10^{-8} for double).

[See `diff_float.c` for a program to verify this—note the truncation error is fairly predictable, but the rounding error is "anywhere in an error-bar"]

Illustration—differentiation (5)

The `Standard_ML` example in Wikipedia quotes an alternative form of differentiation:

$$(f(x+h) - f(x-h))/2h$$

But, applying Taylor's approximation as above to this function gives a truncation error of

$$h^2 f'''(x)/3!$$

The rounding error remains at about $macheps/h$. Now, equating truncation and rounding error as above means that say $h = \sqrt[3]{macheps}$ is a good choice for h.

Entertainingly, until 2007, the article mistakenly said "square root" (. . . recalling the result but applying it to the wrong problem)!

Illustration—differentiation (6)

There is a serious point in discussing the two methods of differentiation as it illustrates an important concept.

Usually when finitely approximating some limiting process there is a number like h which is small, or a number n which is large. Sometimes both logically occur (with $h = 1/n$)

$$\int_0^1 f(x) \approx \left(\sum_{i=1}^{n} f(i/n) \right) /n$$

Often there are multiple algorithms which mathematically have the same limit (see the two differentiation examples above), but which have different *rates of approaching the limit* (in addition to possibly different rounding error accumulation which we're not considering at the moment).

Illustration—differentiation (7)

The way in which *truncation error* is affected by reducing h (or increasing n) is called the *order* of the algorithm (or more precisely the mathematics on which the algorithm is based).

For example, $(f(x+h) - f(x))/h$ is a first-order method of approximating derivatives of smooth function (halving h halves the truncation error.)

On the other hand, $(f(x+h) - f(x-h))/2h$ is a second-order method—halving h divides the truncation error by 4.

A large amount of effort over the past 50 years has been invested in finding techniques which give higher-order methods so a relatively large h can be used without incurring excessive truncation error (and incidentally, this often reduce rounding error too).

Don't assume you can outguess the experts.

Iteration and when to stop

Consider the "golden ratio" $\phi = (1 + \sqrt{5})/2 \approx 1.618$. It satisfies $\phi^2 = \phi + 1$.

So, supposing we didn't know how to solve quadratics, we could re-write this as "ϕ is the solution of $x = \sqrt{x + 1}$".

Numerically, we could start with $x_0 = 2$ (say) and then set $x_{n+1} = \sqrt{x_n + 1}$, and watch how the x_n evolve (an iteration) ...

Golden ratio iteration

i	x	err	err/prev.err
1	1.732050807568877 2	1.1402e-01	
2	1.652891650281069 5	3.4858e-02	3.0572e-01
3	1.628769980777233 3	1.0736e-02	3.0800e-01
4	1.621348198499394 9	3.3142e-03	3.0870e-01
5	1.619057811969478 5	1.0238e-03	3.0892e-01
	...		
26	1.618033988749914 7	1.9762e-14	3.0690e-01
27	1.618033988749900 9	5.9952e-15	3.0337e-01
28	1.618033988749896 7	1.7764e-15	2.9630e-01
29	1.618033988749895 3	4.4409e-16	2.5000e-01
30	1.618033988749894 9	0.0000e+00	0.0000e+00
31	1.618033988749894 9	0.0000e+00	nan
32	1.618033988749894 9	0.0000e+00	nan

Golden ratio iteration (2)

What we found was fairly typical (at least near a solution). The error ϵ_n (defined to be $x_n - \phi$) reduced by a constant fraction each iteration.

This can be seen by expressing ϵ_{n+1} in terms of ϵ_n:

$$\begin{aligned}
\epsilon_{n+1} &= x_{n+1} - \phi = \sqrt{x_n + 1} - \phi \\
&= \sqrt{\epsilon_n + \phi + 1} - \phi \\
&= \sqrt{\phi + 1}\sqrt{1 + \frac{\epsilon_n}{\phi + 1}} - \phi \\
&\approx \sqrt{\phi + 1}\left(1 + \frac{1}{2}.\frac{\epsilon_n}{\phi + 1}\right) - \phi \qquad \text{(Taylor)} \\
&= \frac{1}{2}.\frac{\epsilon_n}{\sqrt{\phi + 1}} = \frac{\epsilon_n}{2\phi} \qquad (\phi = \sqrt{\phi + 1}) \\
&\approx 0.3\epsilon_n
\end{aligned}$$

Golden ratio iteration (3)

The termination criterion: here we were lucky. We can show mathematically that if $x_n > \phi$ then $\phi \leq x_{n+1} \leq x_n$, so a suitable termination criterion is $x_{n+1} \geq x_n$.

In general we don't have such a nice property, so we terminate when $|x_{n+1} - x_n| < \delta$ for some prescribed threshold δ.

Nomenclature:

When $\epsilon_{n+1} = k\epsilon_n$ we say that the iteration exhibits "first order convergence". For the iteration $x_{n+1} = f(x_n)$ then k is merely $f'(\sigma)$ where σ is the solution of $\sigma = f(\sigma)$ to which iteration is converging.

For the earlier iteration this is $1/2\phi$.

Golden ratio iteration (4)

What if, instead of writing $\phi^2 = \phi + 1$ as the iteration $x_{n+1} = \sqrt{x_n + 1}$, we had instead written it as $x_{n+1} = x_n^2 - 1$?

Putting $g(x) = x^2 - 1$, we have $g'(\phi) = 2\phi \approx 3.236$. Does this mean that the error increases: $\epsilon_{n+1} \approx 3.236\epsilon_n$?

Golden ratio iteration (5)

Yes! This is a bad iteration (magnifies errors).

Putting $x_0 = \phi + 10^{-16}$ computes x_i as below:

i	x	err	err/prev.err
1	1.6180339887498951	2.2204e-16	
2	1.6180339887498958	8.8818e-16	4.0000e+00
3	1.6180339887498978	2.8866e-15	3.2500e+00
4	1.6180339887499045	9.5479e-15	3.3077e+00
5	1.6180339887499260	3.1086e-14	3.2558e+00
6	1.6180339887499957	1.0081e-13	3.2429e+00
7	1.6180339887502213	3.2641e-13	3.2379e+00
	...		
32	3.982899489829472	2.3649e+00	3.8503e+00
33	14.863488189986121	1.3245e+01	5.6009e+00
34	219.9232879817058688	2.1831e+02	1.6482e+01

Iteration issues

- Choose your iteration well.
- Determine a termination criterion.

For real-world equations (possibly in multi-dimensional space) neither of these are easy and it may well be best to consult an expert!

Newton–Raphson

Given an equation of the form $f(x) = 0$ then the Newton-Raphson iteration improves an initial estimate x_0 of the root by repeatedly setting

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

See `http://en.wikipedia.org/wiki/Newton's_method` for geometrical intuition – intersection of a tangent with the x-axis.

Newton–Raphson (2)

Letting σ be the root of $f(x)$ which we hope to converge to, and putting $\epsilon_n = x_n - \sigma$ as usual gives:

$$\begin{aligned}
\epsilon_{n+1} &= x_{n+1} - \sigma = x_n - \frac{f(x_n)}{f'(x_n)} - \sigma = \epsilon_n - \frac{f(x_n)}{f'(x_n)} \\
&= \epsilon_n - \frac{f(\sigma + \epsilon_n)}{f'(\sigma + \epsilon_n)} \\
&= \epsilon_n - \frac{f(\sigma) + \epsilon_n f'(\sigma) + \epsilon_n^2 f''(\sigma)/2 + O(\epsilon_n^3)}{f'(\sigma) + \epsilon_n f''(\sigma) + O(\epsilon_n^2)} \\
&= \epsilon_n - \epsilon_n \frac{f'(\sigma) + \epsilon_n f''(\sigma)/2 + O(\epsilon_n^2)}{f'(\sigma) + \epsilon_n f''(\sigma) + O(\epsilon_n^2)} \\
&\approx \epsilon_n^2 \frac{f''(\sigma)}{2f'(\sigma)} + O(\epsilon_n^3) \qquad \text{(by Taylor expansion)}
\end{aligned}$$

This is *quadratic* convergence.

Newton–Raphson (3)

Pros and Cons:

- Quadratic convergence means that the number of decimal (or binary) digits doubles every iteration
- Problems if we start with $|f'(x_0)|$ being small
- (even possibility of looping)
- Behaves badly for multiple roots

Summing a Taylor series

Various problems can be solved by summing a Taylor series, e.g.

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Mathematically, this is as nice as you can get—it unconditionally converges everywhere. However, computationally things are trickier.

Summing a Taylor series (2)

Trickinesses:

- How many terms? [stopping early gives truncation error]
- Large cancelling intermediate terms can cause loss of precision [hence rounding error]
  e.g. the biggest term in $\sin(15)$ [radians] is over -334864 giving (in single precision float) a result with only 1 sig.fig.
  [See `sinseries_float.c`.]

Summing a Taylor series (3)

Solution:

- Do range reduction—use identities to reduce the argument to the range $[0, \pi/2]$ or even $[0, \pi/4]$. However: this might need a lot of work to make $\sin(10^{40})$ or $\sin(2^{100})$ work (since we need π to a large accuracy).
- Now we can choose a fixed number of iterations and unroll the loop (conditional branches can be slow in pipelined architectures), because we're now just evaluating a polynomial.

Summing a Taylor series (4)

If we sum a series up to terms in x^n, i.e. we compute

$$\sum_{i=0}^{i=n} a_i x^i$$

then the first missing term will by in x^{n+1} (or x^{n+2} for $\sin(x)$). This will be the dominant error term (i.e. most of the truncation error), at least for small x.

However, x^{n+1} is unpleasant – it is very very small near the origin but its maximum near the ends of the input range can be thousands of times bigger.

Summing a Taylor series (5)

There's amazing (but non-examinable) technology "Chebyshev polynomials" whereby the total error is re-distributed from the edges of the input range to throughout the input range and at the same time the maximum absolute error is reduced by orders of magnitude.

Basically we merely calculate a new polynomial

$$\sum_{i=0}^{i=n} a'_i x^i$$

where a'_i is a small adjustment of a_i above. This is called 'power series economisation' because it can cut the number of terms (and hence the execution time) needed for a given Taylor series to a produce given accuracy.

Summing a Taylor series (6)

See also the Matlab file `poly.m` from the course web-site about how careless calculation of polynomials can cause noise from rounding error.

(Insert graph here.)

Why fuss about Taylor . . .

. . . and not (say) integration or similar?

It's a general metaphor—in six lectures I can't tell you 50 years of maths and *really clever* tricks (and there is comparable technology for integration, differential equations etc.!). [Remember when the CS Diploma here started in 1953 almost all of the course material would have been on such numerical programming; in earlier days this course would have been called an introduction to "Numerical Analysis".]

But I can warn you that if you need precision, or speed, or just to show your algorithm is producing a mathematically justifiable answer then you may (and will probably if the problem is non-trivial) need to consult an expert, or buy in a package with certified performance (e.g. NAGLIB, Matlab, Maple, Mathematica, REDUCE . . .).

How accurate do you want to be?

If you want to implement (say) $\sin(x)$

```
double sin(double x)
```

with the same rules as the IEEE basic operations (the result must be the nearest IEEE representable number to the mathematical result when treating the argument as precise) then this can require a truly Herculean effort. (You'll certainly need to do much of its internal computation in higher precision than its result.)

On the other hand, if you just want a function which has *known* error properties (e.g. correct apart from the last 2 sig.figs.) and you may not mind oddities (e.g. your implementation of sine not being monotonic in the first quadrant) then the techniques here suffice.

How accurate do you want to be? (2)

Sometimes, e.g. writing a video game, profiling may show that the time taken in some floating point routine like `sqrt` may be slowing down the number of frames per second below what you would like, Then, and only then, you could consider alternatives, e.g. rewriting your code to avoid using `sqrt` or replacing calls to the system provided (perhaps accurate and slow) routine with calls to a faster but less accurate one.

How accurate do you want to be? (3)

Chris Lomont
(`http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf`) writes [fun, not examinable]:

> "Computing reciprocal square roots is necessary in many applications, such as vector normalization in video games. Often, some loss of precision is acceptable for a large increase in speed."

He wanted to get the frame rate up in a video game and considers:

How accurate do you want to be? (4)


```
float InvSqrt(float x){
  float xhalf = 0.5f*x;
  int i = *(int*)&x;          // get bits for floating value
  i = 0x5f3759df - (i>>1); // hack giving initial guess y0
  x = *(float*)&i;            // convert bits back to float
  x = x*(1.5f-xhalf*x*x);  // Newton step, repeat for more accuracy
  return x;}
```

His testing in Visual C++.NET showed the code above to be roughly 4 times faster than the naive `(float)(1.0/sqrt(x))`, and the maximum relative error over all floating point numbers was 0.00175228. [Invisible in game graphics, but clearly numerically nothing to write home about.]

Moral: sometimes floating-point knowledge can rescue speed problems.

Why do I use `float` so much . . .

. . . and is it recommended? [No!]

I use single precision because the maths uses smaller numbers (2^{23} instead of 2^{52}) and so I can use double precision for comparison—I can also use smaller programs/numbers to exhibit the flaws *inherent* in floating point. But *for most practical problems* I would recommend you use `double` almost exclusively.

Why: smaller errors, often no or little speed penalty.

What's the exception: floating point arrays where the size matters and where (*i*) the accuracy lost in the storing/reloading process is manageable and analysable and (*ii*) the reduced exponent range is not a problem.

Notes for C users

- `float` is very much a second class type like `char` and `short`.
- Constants `1.1` are type `double` unless you ask `1.1f`.
- `floats` are implicitly converted to `doubles` at various points (e.g. for 'vararg' functions like `printf`).
- The ISO/ANSI C standard says that a computation involving only `floats` may be done at type `double`, so `f` and `g` in

```
float f(float x, float y) { return (x+y)+1.0f; }
float g(float x, float y) { float t = (x+y); return t+1.0f; }
```

  may give different results.

So: use `double` rather than `float` whenever possible for language as well as numerical reasons. (Large arrays are really the only thing worth discussing.)

UNIVERSITY OF
CAMBRIDGE

Some statistical remarks

How do errors add up in practice?

During a computation rounding errors will accumulate, and in the worst case will often approach the error bounds we have calculated.

However, remember that IEEE rounding was carefully arranged to be statistically unbiased—so for many programs (and inputs) the errors from each operation behave more like independent random errors of mean zero and standard deviation σ.

So, *often* one finds a k-operations program produces errors of around $macheps.\sqrt{k}$ rather than $macheps.k/2$ (because independent random variables' variances sum).

BEWARE: just because the errors tend to cancel for some inputs does not mean that they will do so for all! Trust *bounds* rather than *experiment*.

UNIVERSITY OF
CAMBRIDGE

Some nastier issues

Ill-conditionedness

Consider solving

$$\begin{aligned} x + 3y &= 17 \\ 2x - y &= 6 \end{aligned} \qquad \text{i.e.} \qquad \begin{pmatrix} 1 & 3 \\ 2 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 17 \\ 6 \end{pmatrix}$$

Multiply first equation (or matrix row) by 2 and subtract giving

$$0x + 7y = 34 - 6$$

Hence $y = 4$ and (so) $x = 5$. Geometrically, this just means finding where the two lines given by $x + 3y = 17$ and $2x - y = 6$ intersect. In this case things are all nice because the first line has slope 3, and the second line slope $-1/2$ and so they are nearly at right angles to each other.

Ill-conditionedness (2)

Remember, in general, that if

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix}$$

Then

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} \begin{pmatrix} p \\ q \end{pmatrix} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

Oh, and look, there's a numerically-suspect calculation of $ad - bc$!

So there are problems if $ad - bc$ is small (not absolutely small, consider $a = b = d = 10^{-10}, c = -10^{-10}$, but relatively small e.g. w.r.t. $a^2 + b^c + c^2 + d^2$). The lines then are *nearly parallel*.

Ill-conditionedness (3)

Here's how to make a nasty case based on the Fibonacci numbers

$$f_1 = f_2 = 1, \quad f_n = f_{n-1} + f_{n-1} \quad \text{for } n > 2$$

which means that taking $a = f_n,\ b = f_{n-1},\ c = f_{n-2},\ d = f_{n-3}$ gives $ad - bc = 1$ (and this '1' only looks (absolute error) harmless, but it *is* nasty in relative terms). So,

$$\begin{pmatrix} 17711 & 10946 \\ 6765 & 4181 \end{pmatrix}^{-1} = \begin{pmatrix} 4181 & -10946 \\ -6765 & 17711 \end{pmatrix}$$

and this all looks so harmless, but..

Ill-conditionedness (4)

Consider the harmless-looking

$$\begin{pmatrix} 1.7711 & 1.0946 \\ 0.6765 & 0.4181 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix}$$

Solving we get

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 41810000 & -109460000 \\ -67650000 & 177110000 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

which no longer looks so harmless—as a change in p or q by a very small absolute error gives a huge absolute error in x and y. And a change of a,b,c or d by one in the last decimal place changes any of the numbers in its inverse by a factor of at least 2. (Consider this geometrically.)

Ill-conditionedness (5)

So, what's the message? [Definition of 'ill-conditioned']

This is not just a numerical problem (which it would be if we knew that the inputs were infinitely accurate). The problem is that the solution (x, y) is excessively dependent on small variations (these may arise from measurement error, or rounding or truncation error from previous calculations) on the values of the inputs (a,b,c,d,p and q). Such systems are called *ill-conditioned*. This appears most simply in such matrices but is a problem for many real-life situations (e.g. weather forecasting, global warming models).

A sound approach is to form, or to calculate a bound for, (partial) derivatives of the outputs w.r.t. the inputs $\frac{\partial x}{\partial a}, \ldots, \frac{\partial x}{\partial q}, \frac{\partial y}{\partial a}, \ldots, \frac{\partial y}{\partial q}$ near the point in question. [But this may not be easy!]

Ill-conditionedness (6)

E.g.

$$\frac{\partial}{\partial a}\left\{\frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}\right\} = \frac{-d}{(ad-bc)^2}\begin{pmatrix} d & -b \\ -c & a-(ad-bc)/d \end{pmatrix}$$

Note that uncertainties in the coefficients of the inverse are divided by $(ad-bc)^2$ (which is itself at additional risk from loss of significance).

The problem gets drastically worse as the size of the matrix increases (see next slide).

Ill-conditionedness (7)

E.g. Matlab given a singular matrix finds (rounding error) a spurious inverse (but at least it's professional enough to note this):

```
A = [[16    3     2    13]
     [5    10    11     8]
     [9     6     7    12]
     [4    15    14     1]];
>> inv(A)
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.796086e-18.
ans = 1.0e+15 *
                0.1251    0.3753   -0.3753   -0.1251
               -0.3753   -1.1259    1.1259    0.3753
                0.3753    1.1259   -1.1259   -0.3753
               -0.1251   -0.3753    0.3753    0.1251
```

Note the 10^{15} !!

Ill-conditionedness (8)

There's more theory around, but it's worth noting one definition: the *condition number* $K_f(x)$ of a function f at point x is the ratio between (small) *relative* changes of parameter x and corresponding relative change of $f(x)$. High numbers mean 'ill-conditioned' in that input errors are magnified by f.

A general sanity principle for all maths routines/libraries/packages:

> Substitute the answers back in the original problem and see to what extent they are a real solution. [Didn't you always get told to do this when using a calculator at school?]

Monte Carlo techniques and Ill-conditionedness

If formal methods are inappropriate for determining conditionedness of a problem, then one can always resort to Monte Carlo (probabilistic) techniques.

Take the original problem and solve.

Then take many variants of the problem, each varying the value of one or more parameters or input variables by a few ulps or a few percent. Solve all these.

If these all give similar solutions then the original problem is likely to be well-conditioned. If not, then, you have at least been warned of the instability.

Adaptive Methods

I'm only mentioning these because they are in the syllabus, but I'm regarding them as non-examinable this year.

Sometimes a problem is well-behaved in some regions but behaves badly in another.

Then the best way might be to discretise the problem into small blocks which has finer discretisation in problematic areas (for accuracy) but larger in the rest (for speed).

Sometimes an iterative solution to a differential equation (e.g. to $\nabla^2\phi = 0$) is fastest solved by solving for a coarse discretisation (mesh) and then refining.

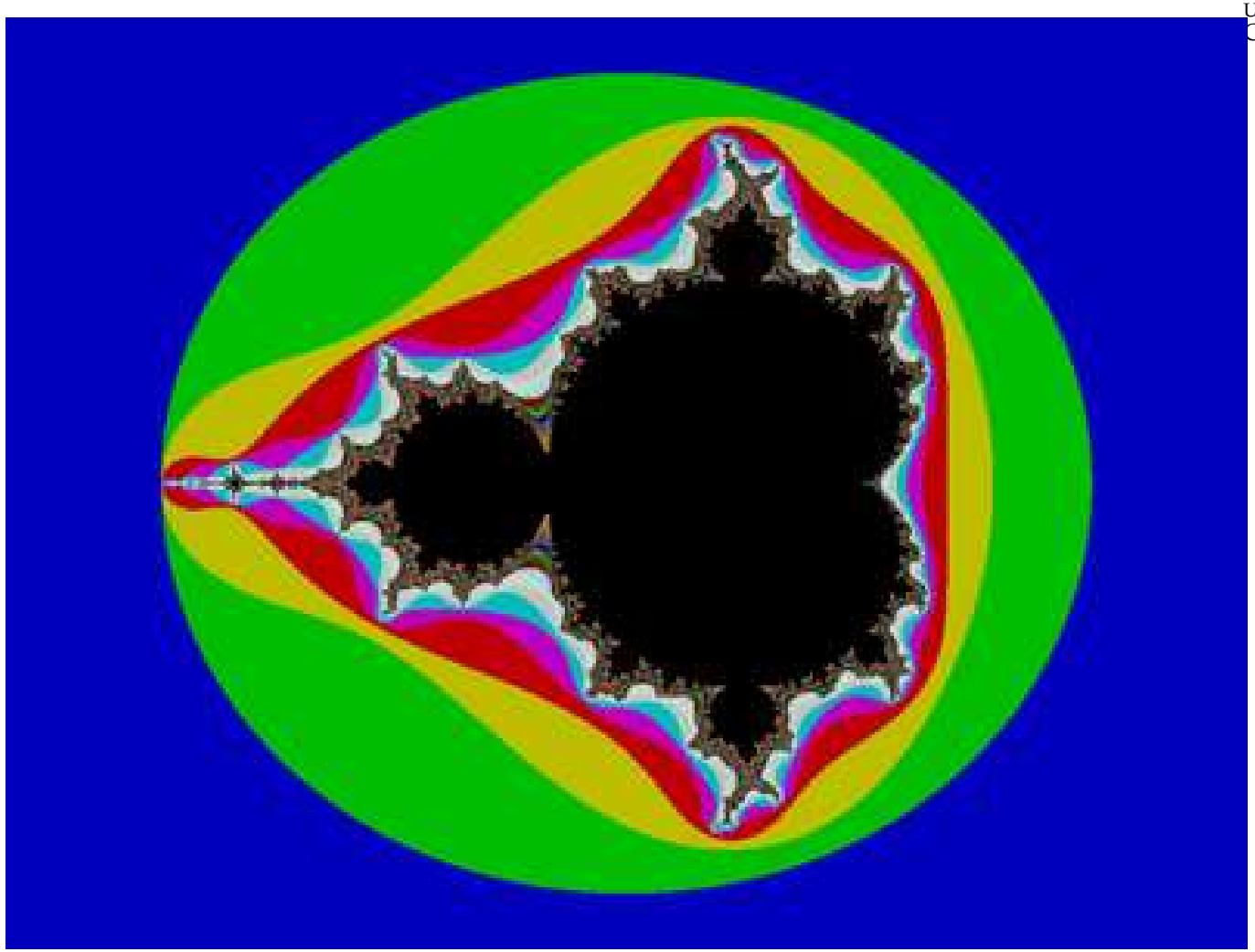
Both these are called "Adaptive Methods".

Chaotic Systems

Chaotic Systems [`http://en.wikipedia.org/wiki/Chaos_theory`] are just a nastier form of ill-conditionedness for which the computed function is highly discontinuous. Typically there are arbitrarily small input regions for which a wide range of output values occur. E.g.

- Mandelbrot set, here we count the number of iterations, $k \in [0..\infty]$, of $z_0 = 0,\ z_{n+1} = z_n^2 + c$ needed to make $|z_k| \geq 2$ for each point c in the complex plane.
- Verhulst's Logistic map $x_{n+1} = r x_n (1 - x_n)$ with $r = 4$
  [See `http://en.wikipedia.org/wiki/Logistic_map` for why this is relevant to a population of rabbits and foxes, and for r big enough (4.0 suffices) we get chaotic behaviour. See later.]

Mandelbrot set



UNIVERSITY OF
CAMBRIDGE

Alternative Technologies to Floating Point

(which avoid doing all this analysis, but which might have other problems)

Alternatives to IEEE arithmetic

What if, for one reason or another:

- we cannot find a way to compute a good approximation to the exact answer of a problem, or
- we know an algorithm, but are unsure as to how errors propagate so that the answer may well be useless.

Alternatives:

- `print(random())` [well at least it's faster than spending a long time producing the wrong answer, and it's intellectually honest.]
- interval arithmetic
- arbitrary precision arithmetic
- exact real arithmetic

Interval arithmetic

The idea here is to represent a mathematical real number value with *two* IEEE floating point numbers. One gives a representable number guaranteed to be lower or equal to the mathematical value, and the other greater or equal. Each constant or operation must preserve this property (e.g. $(a^L, a^U) - (b^L, b^U) = (a^L - b^U, a^U - b^L)$ and you might need to mess with IEEE rounding modes to make this work; similarly 1.0 will be represented as (1.0,1.0) but 0.1 will have distinct lower and upper limits.

This can be a neat solution. Upsides:

- naturally copes with uncertainty in input values
- IEEE arithmetic rounding modes (to +ve/-ve infinity) do much of the work.

Interval arithmetic (2)

This can be a neat solution to some problems. Downsides:

- Can be slow (but correctness is more important than speed)
- Some algorithms converge in practice (like Newton-Raphson) while the computed bounds after doing the algorithm can be spuriously far apart.
- Need a bit more work that you would expect if the range of the denominator in a division includes 0, since the output range then includes infinity (but it still can be seen as a single range).
- Conditions (like $x < y$) can be both true *and* false.

Interval arithmetic (3)

[For Part IB students only]

C++ fans: this is an ideal class for you to write:

```
class interval
{   interval(char *)  { /* constructor... */ }
    static interval operator +(interval x, interval y) { ... };
};
```

and a bit of trickery such as `#define float interval` will get you started coding easily.

Arbitrary Precision Floating Point

Some packages allow you to set the precision on a run-by-run basis. E.g. use 50 sig.fig. today.

For any *fixed* precision the same problems arise as with IEEE 32- and 64-bit arithmetic, but at a different point, so it can be worth doing this for comparison.

Some packages even allow allow *adaptive* precision. Cf. lazy evaluation: if I need $e_1 - e_2$ to 50 sig.fig. then calculate e_1 and e_2 to 50 sig.fig. If these reinforce then all is OK, but if they cancel then calculate e_1 and e_2 to more accuracy and repeat. There's a problem here with zero though. Consider calculating

$$\frac{1}{\sqrt{2}} - \sin(\tan^{-1}(1))$$

(This problem of when an algebraic expression really is exactly zero is formally *uncomputable*—CST Part IB has lectures on computability.)

Exact Real Arithmetic

This term generally refers to better representations than digit streams (because it is impossible to print the first digit (0 or 1) of the sum $0.333333\cdots + 0.666666\cdots$ without evaluating until a digit appears in the sum which is not equal to 9).

This slide is a bit vestigial – I'd originally planned on mentioning to Part IB the (research-level) ideas of using (infinite sequences of linear maps; continued fraction expansions; infinite compositions of linear fractional transformations) but this is inappropriate to Part IA

Exact Real Arithmetic (2)

Results of Verhulst's Logistic map with $r = 4$ (this is a chaotic function) by Martin Plume (Edinburgh):

Iteration	Single Precision	Double Precision	Correct Result
1	0.881836	0.881836	0.881836
5	0.384327	0.384327	0.384327
10	0.313034	0.313037	0.313037
15	0.022702	0.022736	0.022736
20	0.983813	0.982892	0.982892
25	0.652837	0.757549	0.757549
30	0.934927	0.481445	0.481445
40	0.057696	0.024008	0.024009
50	0.042174	0.629402	0.625028
60	0.934518	0.757154	0.315445

Correct digits are underlined (note how quickly they disappear).

Pi to a trillion decimal places

Yasumasa Kanada (Tokyo) exploits formula

$$\pi = 48\tan^{-1}\left(\frac{1}{49}\right)+128\tan^{-1}\left(\frac{1}{57}\right)-20\tan^{-1}\left(\frac{1}{239}\right)+48\tan^{-1}\left(\frac{1}{110443}\right)$$

with

$$\tan^{-1}(x) = \frac{x}{1} - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

and an 80,000 line program to get the current (end-2007) world record for π digit calculation (1.2 trillion decimal places of accuracy).

UNIVERSITY OF
CAMBRIDGE

What have we learned?

What have we learned?

Any floating point computation should be treated with the utmost suspicion unless you can argue how accurate it is

. . . because . . .

here be dragons!