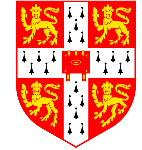
Floating Point Computation

(A four-lecture course)

Alan Mycroft
Computer Laboratory, Cambridge University
`http://www.cl.cam.ac.uk/users/am/`

Michaelmas 2006

Overall motto: threat minimisation

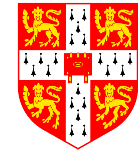


- Algorithms involving floating point (**`float`** and **`double`** in C, [misleadingly named] **`real`** in ML and Fortran) pose a significant threat to the programmer or user.
- Learn to distrust your own naïve coding of such algorithms, and, even more so, get to distrust others'.
- Start to think 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 4 lectures from me.



Introduction/reminding you what you already know

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

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

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.

$$5.67 \times 10^{-5} * 2.34 \times 10^{2} \approx 13.3 \times 10^{-3} = 1.33 \times 10^{-2}$$

$$2.34 \times 10^{2}/5.67 \times 10^{-5} \approx 0.413 \times 10^{7} = 4.13 \times 10^{6}$$

Back to school (2a)

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.

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

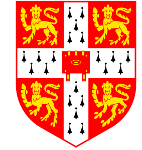
E.g.

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

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.

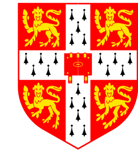


Sex, lies and sig.figs.

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 *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 are quite close, and both have 3sf, but changing the lsd (least significant digit) changes the value by nearly 1% (1 part in 101) in the former and about 0.1% (1 part in 998) in the latter.



Sex, lies and sig.figs.(2)

You might prefer to say $\mathit{sig.figs.}(4.56) = -\log_{10} 0.01/4.56$ so that $\mathit{sf}(1.01)$ and $\mathit{sf}(101)$ is about 3, and $\mathit{sf}(9.98)$ and $\mathit{sf}(0.0000998)$ 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)

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

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) and 7/8sf displayed [Why does it display 8sf for the first number but only 7sf for the rest? I don't know—perhaps just an ordinary bug].

Stress test it:

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

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

Computer Representation

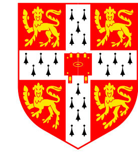
A computer representation must be finite. *If* we allocate a fixed size of storage for each then we need to

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

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 values in scientific notation held in a computer.

But it's now time to turn to binary representations.



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 base = 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.]

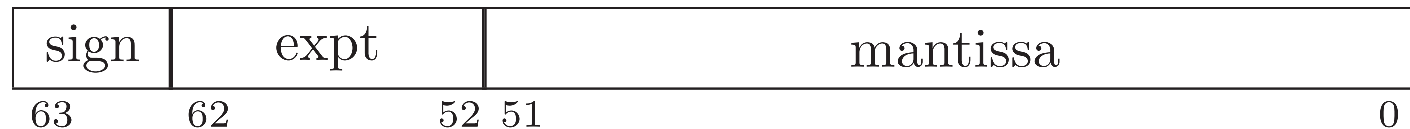
IEEE 854 is more general and allows binary and decimal representation without fixing the bit-level format.

IEEE 754 is being revised –
`http://en.wikipedia.org/wiki/IEEE_754r`

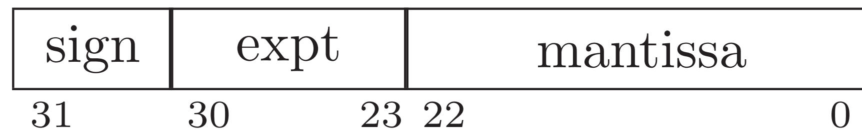
IEEE 754 Floating Point Representation

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), IEEE write $p = 53$



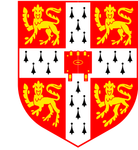
Single precision: 32 bits (1+8+23), IEEE write $p = 24$



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 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)

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: $-3.40282347e+38$! 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 $2^{-150} \approx$ 1.401298e-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 1.79769313486231571e+308 and DBL_MIN 2.22507385850720138e-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 single precision floating point numbers?

Answer: $\pm\, i/2^j$ where $0 \le i < 2^{23}$ and $-126 \le j \le 127$

Compare: what values are exactly representable in 3sf decimal?

Answer: $i/10^j$ where $0 \le i < 1000$.

How many sig.figs. to 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 lsb change is 1 part in 10^8 and $10^7 \le 2^{24} \le 10^8$.

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

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 an 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
```

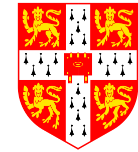
Why infinities and NaNs?

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 given 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 it) and library calls like `sqrt(-1)` generally also return NaNs.



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) lsb.

[This last rule is statistically fairer than the "round down 0–4, round up 5–9" which you learned in school.]

This is a very important slide.

[(*) See next slide]

IEEE Rounding

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

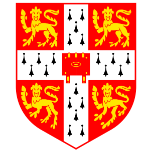
Unbiased which rounds to the nearest value, if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit. This mode is required to be default.

Towards zero

Towards positive infinity

Towards negative infinity

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



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 $\eta = \frac{|a - b|}{|a|}$

[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 = 1.0/9.0
for (i=0; i<30; i++)
{   printf("%e\n", x);
    x = (x - 1) * 10;   // C treats as (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.

Machine Epsilon

Machine epsilon is often defined (e.g. ISO C) 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.

As such it gives an upper bound on the relative error caused by getting the lsb of a floating point number out by one, and is therefore useful for expressing errors independent of floating point size.

In C, using IEEE arithmetic, it is defined as

```
#define FLT_EPSILON         1.19209290e-7F
#define DBL_EPSILON         2.2204460492503131e-16
```

Some (older?) sources define machine epsilon as the smallest number which when added to one gives a number greater than one. (This definition is approximately half of the previous one due to rounding.)

There seems some dispute here, so fine details are non-examinable.

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. (Changing the lsb of the mantissa by one results in an absolute error of $10^{40} \times machine_epsilon$.) Even for `double`, 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?

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 grows over many iterations so as to render the result useless.

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

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

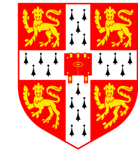
Or in C:

```
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

To be written.



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 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.

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

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

BTW, the Wikipedia entry `http://en.wikipedia.org/wiki/Standard_ML` shows (Nov 2006) a failure to consider x being large or small without noting this fact.

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 $h/f''(x)/2$ (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*):

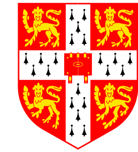
$$(f(x+h)-f(x))/h = (f(x)+hf'(x)\pm macheps-f(x))/h = 1\pm macheps/h$$

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

Equating rounding and truncation errors gives $h = macheps/h$, i.e. $h = \sqrt{machine_epsilon}$ (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"]

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.

Summming 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, 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.

Why fuss about Taylor . . .

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

It's a general metaphor—in four 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?

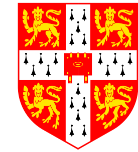
If you want to implement (say) $\sin(x)$

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

with the same rules as the IEEE basic operations (the result must be the nearest IEEE representable number to the 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.

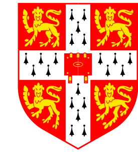
How accurate do you want to be? (2)



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.

Give ACR's nice example code here.

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}), but *for most practical problems* I would recommend you use `double` almost exclusively.

Why: smaller errors, often no little speed penalty.

What's the exception: floating point arrays where the size matters and where the accuracy lost in the storing/reloading process is manageable and analysable.

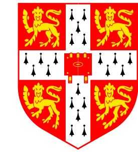
Notes for C users

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

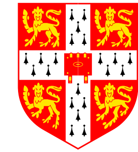
  may give different results.

So: use `double` rather than `float` whenever possible for language as well as numerical reasons.



Some nastier issues

Ill-conditioned and chaotic systems



To be written.

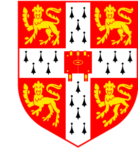
- Ill-conditioned 2x2 matrix equation
- Example Verhulst's Logistic map

$$x_{n+1} = rx_n(1 - x_n)$$

  with $r = 4$.

- Mandelbrot at high magnification?

Part 6



Alternative Technologies to Floating Point (which avoid doing all this analysis)

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 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.

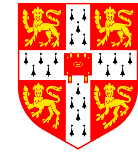
Interval arithmetic (2)

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

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

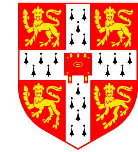
and a bit of trickery such as `#define float interval` will get you started coding easily.

Arbitrary Precision Floating Point



To be written.

Exact Real Arithmetic



To be written.