Computational Projects

Lecture 3: Numbers and numerical errors

Dr Rob Jack, DAMTP

http://www.maths.cam.ac.uk/undergrad/catam/part-ia-lectures

Outline

- How does the computer store and manipulate numbers? (and implications for numerical computation)
- How to quantify the uncertainty in numerical results (important for all computational projects)

Bits

Almost all modern computers store information using binary digits, which are called *bits*

Within the computer, a bit is a physical device that has two possible states (think of a switch that can be on or off)

[Practical bits: small magnets (hard disks), stored electric charge (most computer memory), magnetic tape...]

Bits

Any positive integer x can be represented as

$$x = \sum_{i=0}^{n} a_i \times 2^i$$

for some integers $(n, a_0, a_1, a_2, \ldots, a_n)$, with $a_i \in \{0, 1\}$.

The a_i form a "binary string", for example 14 is represented as 1110, that means $n = 3, a_3 = 1, a_2 = 1, a_1 = 1, a_0 = 0$

Typically one works at fixed n, and each a_i is a bit

To allow for negative numbers let

$$x = -2^n + \sum_{i=0}^{n} a_i \times 2^i$$

Bits

Computers can deal efficiently with numbers for which n is not too large: older computers $n_{\max} = 31$, newer computers $n_{\max} = 63$. "32-bit" or "64-bit" architectures

Integer computations are efficient for numbers x with

$$|x| < 2^{n_{\max}-1}$$

On 64-bit architectures, we are basically ok for integers of up to ~20 digits (in decimal notation). With 32-bits, ok up to ~10 digits.

Real numbers

We often want to work with real numbers,

To do this efficiently, the computer represents these numbers using binary strings of length ℓ (equal to 32 or 64)

This means that of all the real numbers, the computer can work efficiently only with a subset of cardinality 2^ℓ

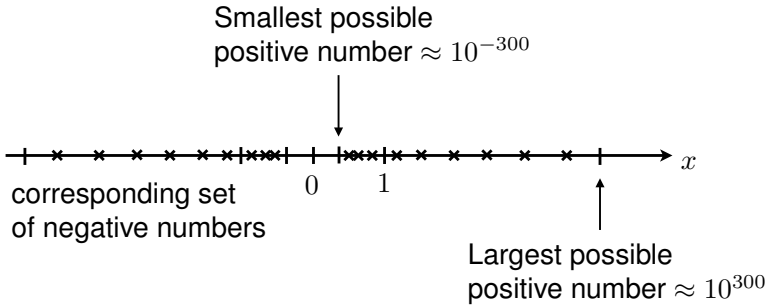
... this is usually not a problem but we have to keep it in mind...

What subset does the computer use, in practice?
(we call this the "set of representable numbers")

Real numbers

What would be a good subset?

[schematic picture not to scale...]



closely-spaced numbers for small x,
wider spacing for larger x

Floating point representation

To turn the picture into mathematics, consider the real number

$$x = a \times 10^b$$

where a, b are integers. For example

$$37.51 = 3751 \times 10^{-2}$$

Eg, for positive numbers, take $A_1 \le a \le A_2$ and $-B_- \le b < B_+$.

This allows for a good range of numbers: the largest possible number is $A_2 \times 10^{B_+ - 1}$), the smallest possible number is $A_1 \times 10^{-B_-}$.

The idea is that for any real number y in the usable range, there should be a representable number x such that $\frac{|x-y|}{|y|}$ is small (eg no larger than $10/A_2$).

Floating point representation

In practice the computer uses

$$x = a \times 2^b$$

where a, b are integers.

For "double precision" (a common choice), a is a 52-bit integer (up to ≈ 16 decimal digits) and b is $\approx$ 11 bits (values up to $\approx 10^3$)

This means that if we have two real numbers x, y with $\frac{|x-y|}{y} \lesssim 10^{-16}$, then *the computer can't resolve the difference between these numbers*.

. . . no exact representation of $\frac{1}{3}, \sqrt{2}, \pi$, etc

. . . root-finding can't work with a tolerance less than ϵ (or at least, it's more complicated. . .)

Machine precision

Let x be the smallest representable number that is (strictly) greater than 1

Then the machine precision is a real number ϵ defined as

$$\epsilon = x - 1$$

```
>> format long
>> eps('double')
ans = 2.220446049250313e-16
>> 2^(-52)
ans = 2.220446049250313e-16
```

Here, 'double' means double precision, which is the standard representation in matlab

With the definitions above, we expect ϵ to be close to $1/A_2$

Absolute vs. relative

Suppose, x is a number of interest

x_c is the number stored by the computer

then,

$x_c - x$ is the *absolute error* of x_c

$\frac{x_c - x}{x}$ is the *relative error* of x_c

Errors that originate in the machine precision are called "round-off" errors.

Since ϵ is very small, perhaps we don't have to worry about it (unless we care about the 16th decimal place. . .)

Two examples of things that can go wrong...

Example 1 -- Quadratic equation

We know that if $x^2 - 2Ax + b = 0$ then $x = A \pm \sqrt{A^2 - b}$

Suppose we are given A, b and we want to compute x.

The number $\sqrt{A^2 - b}$ is likely not representable

. . . we have to approximate it by some representable number C

Then we estimate the roots as $x_1 = A + C$ and $x_2 = A - C$.

Suppose that C has relative error δ. Then the relative errors on x_1 and x_2 can be shown to be:
$\frac{\sqrt{A^2-b}}{A+\sqrt{A^2-b}}\delta$ and $\frac{\sqrt{A^2-b}}{A-\sqrt{A^2-b}}\delta$ respectively

For $\delta = 10^{-7}$, $A = 1000$, $b = 1$, these relative errors are (approx)
5×10^{-8} (on x_1) and 0.2 (on x_2(!))

Example 1 -- Quadratic equation

Result: If we compute $\sqrt{A^2 - b}$ with an accuracy of 6 digits ($\delta = 10^{-7}$), our estimate of x_2 already has errors in the 2nd digit
(relative error ≈ 0.2)

This is a dangerous situation and shows that we must be careful!

The problem usually arises if we have to subtract two similar numbers to get a much smaller number (the spacing between representable numbers is large when the numbers themselves are large)

With good planning it is sometimes possible to rearrange equations to avoid this problem, eg

$$x_2 = A - \sqrt{A^2 - b} = \left(A - \sqrt{A^2 - b}\right) \times \frac{A + \sqrt{A^2 - b}}{A + \sqrt{A^2 - b}} = \frac{b}{A + \sqrt{A^2 - b}}$$

We can estimate this as $\frac{b}{A+C}$, which reduces the relative error.

Example 2 -- Difference equation

Consider the difference equation

$$a_{n+1} = a_{n-1} - a_n$$

Its exact general solution is

$$a_n = \alpha \left(\frac{\sqrt{5}-1}{2}\right)^n + \beta \left(\frac{-\sqrt{5}-1}{2}\right)^n$$

where α, β are constants that depend on a_0 and a_1

Suppose initial conditions are $a_0 = 1$ & $a_1 = \frac{1}{2}(\sqrt{5} - 1)$

Then $\alpha = 1$ and $\beta = 0$ so $a_n = 2^{-n}(\sqrt{5} - 1)^n$

Difference equation

```
% compute and store n iterations of a(n) = a(n-2) - a(n-1)
n = 100;
% the sequence is a matrix of size 1 x (n+1)
numericalSeq = zeros(1,n+1);
% set the first two terms
numericalSeq(1) = 1;
numericalSeq(2) = 0.5*(sqrt(5)-1);

for i = 3:n+1
    numericalSeq(i) = numericalSeq(i-2) - numericalSeq(i-1);
end

for i = 1:n+1
    exactSeq(i) = ((sqrt(5)-1)/2.0)^(i-1);
end

% compute a vector with the (absolute) errors
diffSeq = numericalSeq - exactSeq;

% diff is the error, rel is the relative error
disp('** First 5 terms')
for i = 1:5
    disp(['i=' num2str(i) ...
        ' numeric ', num2str(numericalSeq(i)) ...
        ' exact ' num2str(exactSeq(i)) ...
        ' diff ' num2str(diffSeq(i)) ...
        ' rel ' num2str(diffSeq(i) / exactSeq(i))])
end
```

Example: `diffEq.m`

Difference equation

Result of numerical experiment: the difference between exact and computed sequences starts small but grows exponentially

Also: if we compute an *exact* sequence but for slightly different initial data, we get similar behaviour to the computed solution

Eg take $a_0 = 1$ and $a_1 = \frac{1+\sqrt{5}}{2} - \delta$

Then $\alpha = 1 - (\delta/\sqrt{5})$ and $\beta = \delta/\sqrt{5}$

(numerical example has $\delta \approx \sqrt{5} \times 10^{-16}$)

We get $$a_n = \boxed{\alpha \left(\frac{\sqrt{5}-1}{2}\right)^n} + \boxed{\beta \left(\frac{-\sqrt{5}-1}{2}\right)^n}$$

shrinks with n

starts small, but absolute value grows with n

Difference equation

$$a_n = \boxed{\alpha \left(\frac{\sqrt{5}-1}{2}\right)^n} + \boxed{\beta \left(\frac{-\sqrt{5}-1}{2}\right)^n}$$

shrinks with n

absolute value grows with n

. . . only one shrinking solution ($\beta = 0$),
large family of growing solutions ($\beta \neq 0$)

Interpretation of numerical experiment:

To obtain the exact (decreasing) solution with $\beta = 0$ we would need to set initial condition $a_1 = (\sqrt{5}-1)/2$, which is not representable. . . so in fact our numerical scheme gives us one of the other solutions

(The numerical solution stays close to the exact one as long as n is not too large.)

Round-off: main points

- Finite precision of floating point numbers can affect calculations, via "rounding errors"
- You need to bear in mind that these errors might propagate and affect your calculation
- Small numerical errors can be amplified (for example) when computing small differences between large numbers, or if the solution to your equation is very sensitive to initial conditions

Truncation error

We now consider a different kind of error, that does not come from finite precision.

Truncation error - type 1

Suppose we want a computational estimate of e^x for some specific value of x

We know that $\mathrm{e}^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

The computer can't do the infinite summation but we can fix some N and estimate e^x as

$$C = \sum_{n=0}^{N} \frac{x^n}{n!}$$

Since we truncate the series after N terms, we can define

$$\delta = \mathrm{e}^x - C = \sum_{n=N+1}^{\infty} \frac{x^n}{n!}$$

as the truncation error

(Computers often do calculations in this way but they typically take N large enough the truncation error is comparable to the round-off error)

Truncation error -- type 2

- Truncation also occurs in approximating derivatives, eg

$$\frac{df}{dx}(x) = \frac{1}{h}[f(x+h) - f(x)] + O(h)$$

- These kinds of approximation are often used when solving ordinary differential equations (see next lecture)

Making a Taylor expansion and rearranging, truncation error is

$$\sum_{n=2}^{\infty} \frac{1}{n!} \frac{\mathrm{d}^n f}{\mathrm{d}x^n}(x) h^{n-1}$$

(assuming all derivatives exist at x and that the sum converges)

... next lecture

numerical solution of ODEs, and associated errors

... likely to be relevant for a part IB core project