Computational Physics
Lecture 2 – Introduction to Numerical Calculations

Pat Scott

Department of Physics, Imperial College

October 5, 2018

Slides available from https://bb.imperial.ac.uk/

Outline

Outline

- **logicals/booleans**
  - Single bit
  - 0/1 = true/false
- **integers**
  - multiple logicals, each corresponding to a power of 2
  - i.e. just natural numbers base 2
  - unsigned or signed (+1 *sign bit*)

$$\begin{array}{cccccccccc} 0 & 0 & 1 & 1 & & 0 & 1 & 0 & 0 & \\ + & 0 & 32 & 16 & & 0 & 4 & 0 & 0 & = 52 \end{array}$$

  - 'fixed' point
  - short (15+1-bit binary strings) or long (31+1)

- **strings/characters**
  - ‘cat’, ’dog’, ‘Physics’, etc.
  - each character is encoded by some set number of bits
  - In standard ASCII, there are 7 bits encoding $2^7 = 128$ characters (not all printable)
  - ‘a’ = 97 (decimal) = 61 (Hexadecimal) = 0110 0001 (binary)
- **pointers**
  - Integer memory addresses saved as variables
  - used to refer to other variables, functions, subroutines, pointers, objects, etc
- **references (mostly relevant to C++)**
  - Like pointers, but more of an alias than a variable
- **arrays, structures, classes, objects, etc**

  Just bundles of other variables, often with additional operations defined on them

- **'floating' point**
  - inexact representation of $\mathbb{R}$
  - sign bit s, exponent E and mantissa/significand M
  - number = $(-1)^s \cdot \beta^{E-e} \cdot M$ (given base β and bias e)

For *this* example
$\beta = 2$, $e = 129$

IEEE754-1985
has $e = 127$

	sign bit	8-bit exponent	23-bit mantissa (this bit could be "phantom")
$1/2$ =	0	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 =	0	1 0 0 0 0 0 1 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$1/4$ =	0	0 1 1 1 1 1 1 1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

  - single ($1 + 23 + 8 = 32$-bit), double ($1 + 52 + 11 = 64$-bit), extended/quadruple ($1 + 112 + 15 = 128$-bit)
- **complex**
  - composite of two floating-point numbers
  - plus some 'interaction terms'

The bible: What Every ~~Computer~~ Scientist Should Know About Floating-Point Arithmetic – Goldberg (1991) (available on Blackboard)

How would you represent -2.5?

How would you represent -2.5?

2 minutes to think about this for yourselves (don't talk to others).

number = $(-1)^s \cdot \beta^{E-e} \cdot M$ (given base β and bias e)

	sign bit	8-bit exponent	this bit could be "phantom"	23-bit mantissa
$1/2$ =	0	1 0 0 0 0 0 0 0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 =	0	1 0 0 0 0 0 1 0	1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$1/4$ =	0	0 1 1 1 1 1 1 1	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(For *this* example $\beta = 2$, $e = 129$)

How would you represent -2.5?

2 minutes to think about this for yourselves (don't talk to others).

2 minutes to discuss with your neighbour.

number = $(-1)^s \cdot \beta^{E-e} \cdot M$ (given base β and bias e)

	sign bit	8-bit exponent	23-bit mantissa (this bit could be "phantom")
$1/2 =$	0	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$3 =$	0	1 0 0 0 0 0 1 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$1/4 =$	0	0 1 1 1 1 1 1 1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(For *this* example $\beta = 2$, $e = 129$)

How would you represent 3.25?

How would you represent 3.25?

2 minutes to think and discuss with your neighbour.

number = $(-1)^s \cdot \beta^{E-e} \cdot M$ (given base β and bias e)

sign bit

8-bit exponent

this bit could be "phantom"

23-bit mantissa

(For *this* example $\beta = 2$, $e = 129$)

	sign bit	8-bit exponent	23-bit mantissa
$\frac{1}{2}$ =	0	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 =	0	1 0 0 0 0 0 1 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$\frac{1}{4}$ =	0	0 1 1 1 1 1 1 1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Floating-point Addition and Subtraction

$A + B$ (when $A > B$)

1. shift B to the same exponent as A
2. integer add/subtract the mantissae

Floating-point Addition and Subtraction

What about when the two numbers are very different?

$\beta = 2$, $e = 129$ again

	sign bit	8-bit exponent	this bit could be "phantom" / 23-bit mantissa
$3 =$	0	1 0 0 0 0 0 1 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$10^{-7} =$	0	0 1 1 0 1 0 0 1	1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 0
$=$	0	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$3 + 10^{-7} =$	0	1 0 0 0 0 0 1 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Two numbers are very different in size $\implies$ the smaller is ignored

- There exists a smallest number that can be added/subtracted meaningfully from 1.0 – machine accuracy

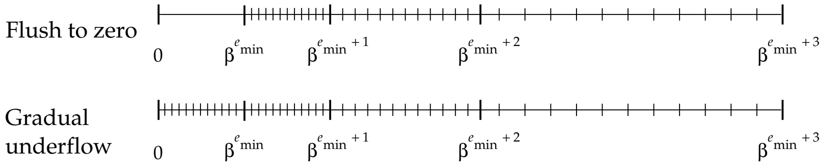
Two numbers very similar size $\implies$ subtraction causes cancellation errors

- Significant digits are lost, only those affected by round-off remain

Floating-point Multiplication & Division

$A \times B$

1. XOR (multiply) the sign bit
2. integer add/subtract the exponents
3. integer multiply/divide the mantissae



Going $<$ smallest *normalisable* number in the rep. causes headaches

- subnormal/denormalised numbers are inaccurate and slow

Rounding/representation error

- multiplies as floats are operated on
- always a problem, but magnified with denormalised numbers

Other limitations

Overflows

- Above the representable range for |`number`|
- Integers tend to wrap around (or throw errors)
- Floats go to $\pm$`Inf`

Underflows

- Only an issue for floats
- number $\rightarrow \pm 0$

NaNs

- Not a Number
- Only really exists in floating-point specification
- Most often results from
    1. a divide by 0
    2. operations on $\pm$`Inf`
    3. imaginary results for real functions

Operations on integers

- Integer operations and arithmetic are *fast* and exact
- Well, except for division
  - in this case the remainder is truncated
  - there are other tricks to get quick rounded results

- Standard logical operations can be performed bitwise on whole integers at once

  This is good for:
  - quick multiplication or division by 2
  - extracting/setting combinations of flags (booleans)

Comparison using floating-point variables

- Never, ever write

  `if (myVar == 0.)` or `if (myVar1 == myVar2)`
- Floating point numbers cannot be trusted to be exactly equal to anything!!!
- Much safer to write

  `if (abs(myVar)< absprec)`

  or

  `if (abs(myVar1/myVar2 - 1.0)< relprec)`

  where `absprec` = $A\epsilon$ and `relprec` = $B\epsilon$
- Floating point comparisons only OK to within some suitable multiple $A, B >> 1$ of ϵ!!!
- In practice A and B are set per-algorithm (according to round-off error, speed, truncation error, etc)

Outline

Renormalisation of variables ('natural units')

- Avoid working with very big floats
  - **Speed** - most CPUs are quicker doing double prec than extended prec
  - **Space** - doubles take less space than extendeds (does matter sometimes)
  - **Accuracy** - it is easy to go beyond the max/min representable number
- Also avoid working with very small floats
  - **Speed** - denormalised arithmetic is slooooow
  - **Space** - as with big floats: prefer lower precision
  - **Accuracy** - denormalised arithmetic is inaccurate
- Instead, choose a scale and renormalise

$$A \to B \equiv A/A_{\max}$$

- Easier + faster to multiply by a scalefactor afterwards, even if that requires type conversion

Renormalisation of variables (‘natural units’)

- Consider working in log space
  - if your data span a huge range of scales, and you care about the small ones
  - e.g. a range of 2–70 is way easier to work with than 100–10^{70}
  - often a lot faster and more accurate; no huge or tiny floats
- Easier + faster to exponentiate afterwards, even if that requires type conversion

Outline

Solving equations

Everybody needs to solve an equation numerically eventually...

$$f(x) + a = g(x) + b$$

Solving equations

Everybody needs to solve an equation numerically eventually...

$$f(x) + a = g(x) + b$$

$$f(x) - g(x) + a - b = 0 \tag{1}$$

$$\text{i.e. } h(x) = 0 \tag{2}$$

Recast it as homogeneous and you have

The classic root-finding problem

For what x does $h(x) = 0$?

First. . .

Guess!

First. . .

Guess!

Then guess again!

First. . .

Guess!

Then guess again!

If your guesses have the same sign for $h(x)$, keep guessing. . .

First. . .

Guess!

Then guess again!

If your guesses have the same sign for $h(x)$, keep guessing. . .

Eventually, you'll get two opposite sign values for $h(x)$. Now you're in business. . .

First. . .

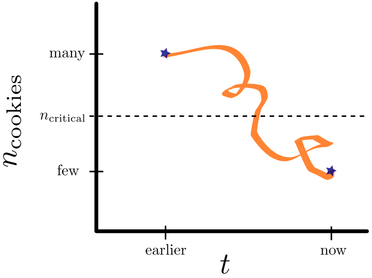
Guess!

Then guess again!

If your guesses have the same sign for $h(x)$, keep guessing. . .

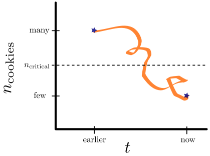
Eventually, you'll get two opposite sign values for $h(x)$. Now you're in business. . .

Intermediate value theorem $\implies$ there must be some root between the guesses



Bracketing

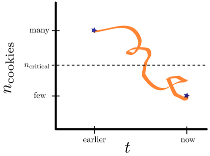
Intermediate value theorem $\implies$ there must be some root between the guesses



- The point of root-finding is to refine these ‘brackets’ as quickly as possible.
- Bracketing is essential.

Bracketing

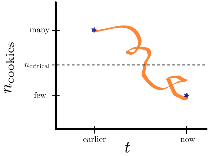
Intermediate value theorem $\implies$ there must be some root between the guesses



- The point of root-finding is to refine these 'brackets' as quickly as possible.
- Bracketing is essential.
- If *all* your guesses have the same sign for $h(x)$, you're a bit screwed – find something better than guessing. Actually, work out how to guess smarter.

Bracketing

Intermediate value theorem $\implies$ there must be some root between the guesses



- The point of root-finding is to refine these 'brackets' as quickly as possible.
- Bracketing is essential.
- If *all* your guesses have the same sign for $h(x)$, you're a bit screwed – find something better than guessing. Actually, work out how to guess smarter.
- Always eyeball your function before trying to find its roots, unless you know it very well.

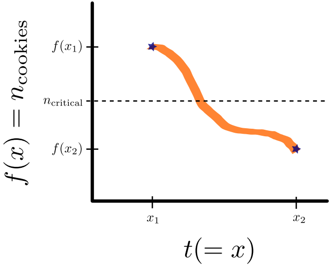
Bisection

Divide and conquer:

Bisection

Divide and conquer:

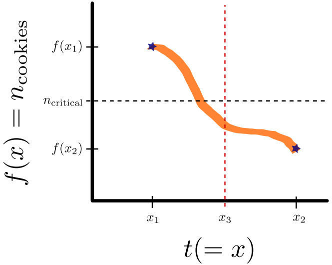
1. Start with a root bracketed by values x_1 and x_2



Bisection

Divide and conquer:

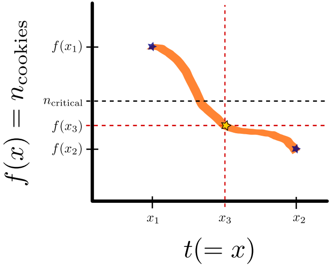
1. Start with a root bracketed by values x_1 and x_2
2. Find the midpoint as $x_3 = \frac{x_1+x_2}{2}$



Bisection

Divide and conquer:

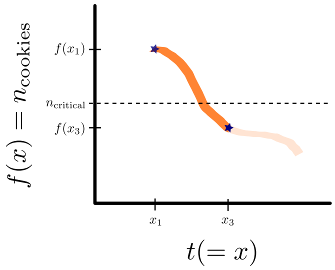
1. Start with a root bracketed by values x_1 and x_2
2. Find the midpoint as $x_3 = \frac{x_1+x_2}{2}$
3. Evaluate $f(x_3)$



Bisection

Divide and conquer:

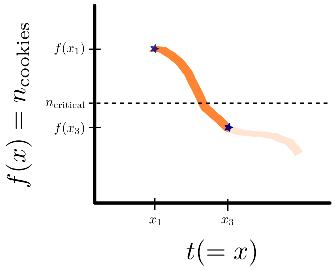
1. Start with a root bracketed by values x_1 and x_2
2. Find the midpoint as $x_3 = \frac{x_1+x_2}{2}$
3. Evaluate $f(x_3)$
4. Discard whichever of x_1 or x_2 gives f the same sign as $f(x_3)$



Bisection

Divide and conquer:

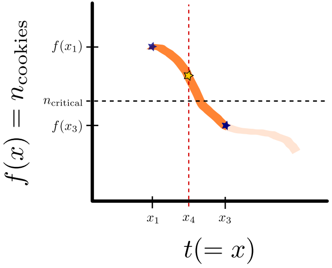
1. Start with a root bracketed by values x_1 and x_2
2. Find the midpoint as $x_3 = \frac{x_1+x_2}{2}$
3. Evaluate $f(x_3)$
4. Discard whichever of x_1 or x_2 gives f the same sign as $f(x_3)$
5. The root is now bracketed by x_3 and the remaining one of x_1 or x_2



Bisection

Divide and conquer:

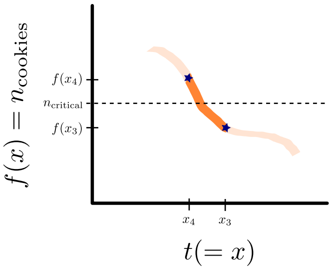
1. Start with a root bracketed by values x_1 and x_2
2. Find the midpoint as $x_3 = \frac{x_1+x_2}{2}$
3. Evaluate $f(x_3)$
4. Discard whichever of x_1 or x_2 gives f the same sign as $f(x_3)$
5. The root is now bracketed by x_3 and the remaining one of x_1 or x_2
6. Repeat



Bisection

Divide and conquer:

1. Start with a root bracketed by values x_1 and x_2
2. Find the midpoint as $x_3 = \frac{x_1+x_2}{2}$
3. Evaluate $f(x_3)$
4. Discard whichever of x_1 or x_2 gives f the same sign as $f(x_3)$
5. The root is now bracketed by x_3 and the remaining one of x_1 or x_2
6. Repeat



Improving on bisection

General idea for improving is to use some (convergent) approximation / guess function

- Linear interpolation $=$ secant, false position method
- Exponential functions $=$ Ridder's method
- Quadratic interpolation ($+$bisection) $=$ Müller's method
- Inverse quadratic interpol ($+$bisection) $=$ Brent's method
- Tangent extrapolation $=$ Newton-Raphson

More info available in the lecture notes.

- Problem Sheet for this lecture is available on Blackboard
  - Floating point issues
  - Variable rescaling / natural units
- Next lecture:
  - 11am Tuesday
  - Matrix Methods

Bonus content: a horror story about default variable widths

- Most new systems are 64-bit
- Most scientific code was written on 32-bit machines
- Variable widths can differ!

Classic nasty bug: passing pointers between C and Fortran

- C pointer has same width as long int (32 bits) on 32-bit, but twice the width on 64-bit
- Fortran has no separate pointer type (attribute only)
- compilers don't care/know $\rightarrow$ linked C-F90 code compiles fine on both machines
- Crashes with seg fault on 64-bit machine, runs OK on 32-bit
- $\implies$ not enough space reserved for 64-bit pointer in C-F90 interface