Numerical Methods

A twelve-lecture course.
Not all topics may be covered - see first section of Learners' Guide.

Dr. D J Greaves
Computer Laboratory, University of Cambridge

(The projected slide pack is online. It contains further minor slides.)
`http://www.cl.cam.ac.uk/teaching/current/NumMethods`

Easter Term 2017/18

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 (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, it only gives small variations in the result when any of the input data is varied slightly.

So far so good.

Interviewer: so you're pretty confident about your modelling progam?

Dr X: Oh yes, indeed it gives results in the same range even if any of the input arguments are randomly permuted . . .

Oh dear!

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 0.2 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 FDTD simulator 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.
- Some well-known and widely-used numerical techniques.
- Knowing when to call in an expert—remember there is 50+ years of knowledge on this and you only get 11 lectures from me.

Part 1

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 a 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 digital signal processing of audio). Note $1111.1111_2 = 15.9375_{10}$.

An alternative way to greatly reduce or 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 part of the 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)

Steps for **multiplication and division:**

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.

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

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

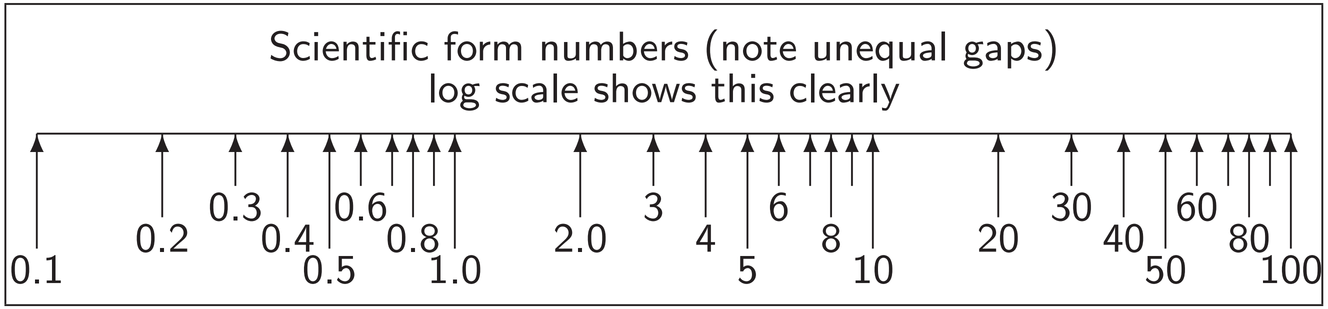
When using scientific-form we often compute repeatedly keeping the same number of digits in the mantissa. In science this is often the number of digits of accuracy in the original inputs—hence the term (decimal) *significant figures* (sig.figs. or sf).

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)

This scale shows the reperesentable numbers in base ten using a mantissa of one digit plotted over three decades. A log scale has been used. **We note the unevenness.** They would still be uneven on a linear scale, but the picture would be different and harder to view.


Scientific form numbers (note unequal gaps)
log scale shows this clearly
0.1
0.2
0.3
0.4
0.5
0.6
0.8
1.0
2.0
3
4
5
6
8
10
20
30
40
50
60
80
100

You might prefer to say *sig.figs.*(4.56) is $\log_{10}(4.56/0.01) \sim 2.7 \sim 3$.

So that $sf(1.01)$ and $sf(101)$ is about 3,

and $sf(9.98)$ and $sf(0.0000998)$ is nearly 4.

Get your calculator out!

Scientific calculators use floating point arithmetic units.

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)

$$(1+1\text{e}20)-1\text{e}20 = 0.000000 \qquad 1+(1\text{e}20-1\text{e}20) = 1.000000$$

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

[This is a demonstration of what we will later define as **underflow**.]

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 simply 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.

Since e will have a fixed-width finite encoding, its range must also be specified.

All contemporary, general purpose, digital machines use binary and keep $\beta = 2$.

For discussion:

1. Discover and explain a rule of thumb that relates the number of digits in a decimal expression of a value with its binary length.
2. Consider whether $(\beta = 2, p = 24)$ is less accurate than $(\beta = 10, p = 10)$ (answer coming up soon . . .).
3. The IBM/360 series of mainframes used $\beta = 16$ which gave some entertaining but obsolete problems. How does its accuracy compare with $\beta = 2$ systems?

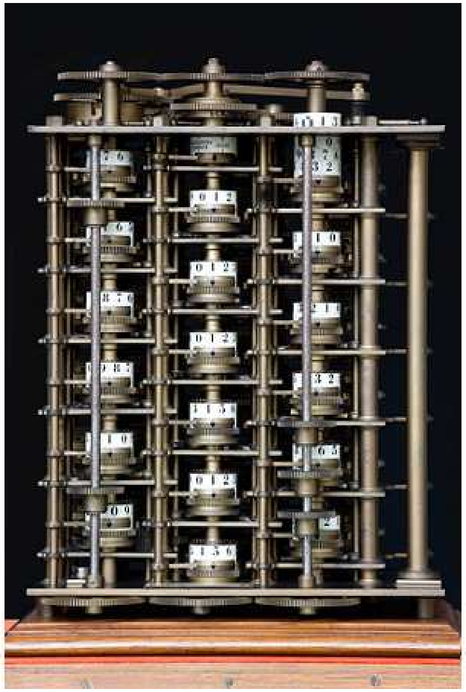
Decimal or Binary?

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, $\beta = 10$, since they generally do as much I/O as internal processing, but various software calculators work in binary (see `xcalc` above).

Charles Babbage's machines also used $\beta = 10$ which made debugging easier: just read the value off the digit wheels in each register!



Aside: Spreadsheet Experiments

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.

Enter the following four items into the four cells named:

```
A1: 1.2E+200      C1: =A1+B1
B1: 1E+100        D1: =IF(A1=C1)
```

So far so good?

Can you adjust A1 or B1 so that C1 is displayed like A1 but D1 is false ?

Long Multiplication - Using Broadside Addition

Long Multiply Radix 2

```
fun mpx1(x, y, c) =
    if x=0 then c else
    let val (x’, n) = (x div 2, x mod 2)
        val y’ = y * 2
        val c’ = case n of
           0  => (c)
         | 1  => (c+y)
    in mpx1(x’, y’, c’) end

fun mpx2(x, y, c, carry) =
    if x=0 andalso carry=0 then c else
    let val (x’, n) = (x div 2, x mod 2 + carry)
        val y’ = y * 2
        val (carry’, c’) = case  n  of
           0  => (0, c)
         | 1  => (0, c+y)
         | 2  => (1, c)
    in mpx2(x’, y’, c’, carry’) end
```

Long Multiply Radix 4

```
fun booth(x, y, c, carry) =
    if x=0 andalso carry=0 then c else
    let val (x’, n) = (x div 4, x mod 4 + carry)
        val y’ = y * 4
        val (carry’, c’) = case n of
           0  => (0, c)
         | 1  => (0, c+y)
         | 2  => (0, c+2*y)
         | 3  => (1, c-y)
         | 4  => (1, c)
     in booth(x’, y’, c’, carry’)
     end

// The mpx2 function does the same as mpx1
// but is styled to look like booth above.
```

Multiplying or dividing by powers of 2 uses shifts which are ‘free’ in binary hardware. Booth’s algorithm avoids multiplying by 3 using a negative carry concept.

Broadside is a hardware term meaning doing all the bits in one operation (or clock cycle).

Exercises: Multidigit (long) multiplication is linear or quadratic in word length? Give the code for base/radix 4 multiplication without Booth’s optimisation. (All exercises shown on slides are also be on the Exercise Sheet(s).)

Standard Long Division Algorithm (Variable Latency)

```
fun divide N D =
    let fun prescale p D = if N>D then prescale (p*2) (D*2) else (p, D)
        val (p, D) = prescale 1 D (* left shift loop *)

        fun mainloop N D p r =
            if p=0 then r       (* could also return remainder from N *)
            else
                (* Binary decision - it either goes or doesn't *)
                let val (N, r) = if N >= D then (N-D, r+p) else (N, r)
                     in mainloop N (D div 2) (p div 2) r
                     end
       in mainloop N D p 0
       end
```

```
val divide = fn: int -> int -> int
divide 1000000 81;
val it = 12345: int
```

Considerations: A hardware implementation that uses one clock cycle per iteration is very simple. Typically we use more silicon and, say, four to eight times fewer clock cycles.

Exercise: Do these 'long' algorithms work with two's complement numbers? In one or both arguments? If not what changes are needed? What happens if we divide by zero?

[See `demos/long-division`.]

Fixed-Latency Long Division Algorithm

```
val NUMBASE = 1073741824 (* Two to the power of 30. *)

(* We use the pair Nh,Nl to form a double-width register. *)

fun divide N D =
     let
        fun divloop (Nh, Nl) p q =
            if p=0 then q
            else
                let val (Nh, Nl) = (Nh*2 + Nl div NUMBASE, (Nl mod NUMBASE)*2) in
                let val (NhNl, q) = if Nh >= D then ((Nh-D, Nl), q+p) else ((Nh, Nl), q)
                     in divloop NhNl (p div 2) q
                     end end
       in divloop (0, N) NUMBASE 0
    end
```

```
val divide = fn: int -> int -> int
divide 1000000 81;
val it = 12345: int
```

Considerations: Is this a different 'algorithm' from the standard algorithm? Which one is quicker on average and in the worst case?

What does the fixed-latency algorithm return for a divide by zero and why?

Numeric Base Conversions

Computers commonly have to convert between base 10 and base 2.

Early computers and many pocket calculators work internally in base 10 - saves I/O errors.

COBOL programming language works in base 10 - avoids lost pennies.

Generally, we need the following four conversions:

1. Integer Input (ASCII Characters to Binary)
2. Integer Output (Binary to ASCII)
3. Floating Point Input - (left as an exercise)
4. Floating Point Output

An asymmetry between the input and algorithms arises owing to the computer typically only having binary arithmetic.

Integer Input (ASCII Characters to Binary)

```
   fun atoi [] c = c
   |   atoi (h::t) c = atoi t (c*10 + ord h - ord #"0")
   ;

val atoi = fn: char list -> int -> int

atoi (explode "12345") 0;
val it = 12345: int
```

Method: Input characters are processed in order. The previous running total is multipled by ten and the new digit added on.

Cost: Linear.

Integer Output (Binary to ASCII)

```
val tens_table = Array.fromList [1,10,100,1000,10000,100000, ...];

fun bin2ascii d0 =
    let fun scanup p = if Array.sub(tens_table, p) > d0 then p-1 else scanup(p+1)

        val p0 = scanup 0
        fun digits d0 p =
            if p<0 then [] else
            let val d = d0 div Array.sub(tens_table, p)
                val r = d0 - d * Array.sub(tens_table, p)
            in chr(48 + d) :: digits r (p-1) end
        in digits d0 p0 end
```

```
bin2ascii 1234;
val it = [#"1", #"2", #"3", #"4"]: char list
```

Integer output: this is a variant of the standard long division algorithm.

If $x < 0$, first print a minus sign and proceed with $0 - x$.

Exercise: Modify this code to avoid supressing all leading zeros in its special case. [See `demos/bin-to-ascii`.]

Floating Point Output: Double Precision to ASCII

If we want, say 4, significant figures:

1. scale the number so it is in the range 1000 to 9999
2. cast it to an integer
3. convert the integer to ASCII as normal, but either corporally insert a decimal point or add a trailing exponent denotation.

We first will need some lookup tables:

```
val f_tens_table = Vector.fromList [1E0,1E1,1E2,1E3,1E4,1E5,1E6,1E7,1E8];

val bin_exps_table =  [ (1.0E32, 32),  (1.0E16, 16),  (1.0E8, 8),
                        (1.0E4, 4),  (1.0E2, 2),  (1.0E1, 1) ];
val bin_fracs_table = [ (1.0E~32, 32), (1.0E~16, 16), (1.0E~8, 8),
                         (1.0E~4, 4), (1.0E~2, 2), (1.0E~1, 1) ];
```

Logarithmically larger tables will be needed for a larger input range.

[See `demos/float-to-ascii`.]

Floating Point Output: Double Precision to ASCII (first half)

```
fun float_to_string precision d00 =
  let val lower_bound = Vector.sub(f_tens_table, precision)
      val upper_bound = Vector.sub(f_tens_table, precision+1)
      val (d0, sign) = if d00 < 0.0 then (0.0-d00, [#"-"]) else (d00, [])
      fun chop_upwards ((ratio, bitpos), (d0, exponent)) =
           let val q = d0 * ratio
           in if q < upper_bound then  (q, exponent - bitpos) else (d0, exponent)
           end

      fun chop_downwards ((ratio, bitpos), (d0, exponent)) =
           let val q = d0 * ratio
           in if q > lower_bound then  (q, exponent + bitpos) else (d0, exponent)
           end

      val (d0, exponent) = if d0 < lower_bound then foldl chop_upwards (d0, 0) bin_exps_table
                                              else foldl chop_downwards (d0, 0) bin_fracs_table

      val imant = floor d0 (* Convert mantissa to integer. *)
```

Floating Point Output: Double Precision to ASCII (second half)

```
val exponent = exponent + precision

(* Decimal point will only move a certain distance: outside that range force scientific form. *)
val scientific_form = exponent > precision orelse exponent < 0

fun digits d0 p trailzero_supress =
     if p<0 orelse (trailzero_supress andalso d0=0) then [] else
     let val d = d0 div Array.sub(tens_table, p)
         val r = d0 - d * Array.sub(tens_table, p)
         val dot_time =  (p = precision + (if scientific_form then 0 else 0-exponent))
         val rest = digits r (p-1) (trailzero_supress orelse dot_time)
         val rest = if dot_time then #"."::rest else rest (* render decimal point *)
         in if d>9 then #"?" :: bin2ascii d0 @ #"!" :: rest else chr(ord(#"0") + d) :: rest end

val mantissa = digits imant precision false
val exponent = if scientific_form then #"e" :: bin2ascii exponent else []
in (d00, imant, implode(sign @ mantissa @ exponent) )
end
```

Floating Point Output: Double Precision to ASCII (4)

The key part is on the first page embodying the central iteration (successive approximation) that scans one of the two lookup tables.

then there is the correction: `exponent += precision;`

finally we have a lightly modified int-to-ASCII render with decimal point.

Let's test it:

```
map (float_to_string 4) [ 1.0, 10.0, 1.23456789, ~2.3E19, 4.5E~19 ];
> val it =
   [(1.0, 10000, "1."),          (10.0, 10000, "10."),          (1.23456789, 12345, "1.2345"),
    (~2.3E19, 23000, "-2.3e19"), (4.5E~19, 45000, "4.5e~19")]
```

Exercise: Explain why this code can handle the range $10^{\pm 63}$. Is this sufficient for single and double precision IEEE?

Ceiling and Floor Functions

Ceiling and floor: convert floating-point to integer rounding up or down.

```
int Ceil(double arg);          // Nearest integer rounding up
int Ceil(float arg);


int Floor(double arg);         // Nearest integer rounding down
int Floor(float arg);


int round(double arg)          // Nearest integer rounding fairly
       { return Floor(arg + 0.5);  }
```

Conversion (promotion) from integer to double is tacitly inferred in most HLLs.

```
int a = 30;
float b = 10.6;
double r = a + b;   // Results in 40.6 in double precision
```

Part 2

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.

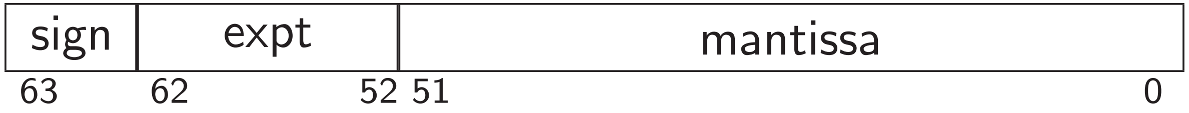
IEEE 754 is a binary standard that requires $\beta = 2, p = 24$ (number of mantissa bits) for *single precision* and $p = 53$ for *double precision*. It also specifies the precise layout of bits in a single and double precision.
[Edited quote from Goldberg.]
In 2008 it was augmented to include additional 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

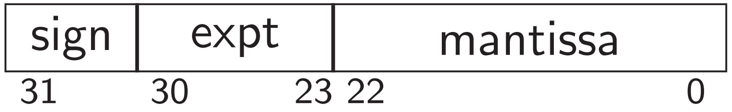
Here I give the version used on x86 (similar issues arise as in the ordering of bytes with an 32-bit integer).

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



sign	expt	mantissa
63	62 52	51 0

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



sign	expt	mantissa
31	30 23	22 0

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.

Like base 10 where 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. Cool!

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 one) 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`.

[See demos/numrange_float.]

Solved exercises (2)

‘denormalised numbers’ can range down to $2^{-149} \approx 1.401298$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 1.79769313486231571e+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 ...).

Solved exercises (3)

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

Legal Answer: $\pm\,((2^{23}+i)/2^{23})\times 2^j$ where $0 \leq i < 2^{23}$ and $-126 \leq j \leq 127$ (because of hidden bit)

More Useful Answer: $\pm\, i \times 2^j$ where $0 \leq i < 2^{24}$ and $-126-23 \leq j \leq 127-23$ (but strictly 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 $100 \leq i \leq 999$.

Solved exercises (4)

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. $\begin{array}{|c|c|c|}\hline 0 & 011\ 1101\ 1 & 100\ 1100\ 1100\ 1100\ 1100\ 1101 \\ \hline\end{array}$.

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’ 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.

Dividing by a Constant: Example divide by ten.

In binary, one tenth is 0.0001100110011....

Divide a 32 bit unsigned integer by 10 by long multiplication by reciprocal using the following hand-crafted code:

```
unsigned div10(unsigned int n)
{ unsigned int q;
  q = (n >> 1) + (n >> 2);  // Ultimately shifts of 4 and 5.
  q = q + (q >> 4);         // Replicate: 0.11 becomes 0.110011
  q = q + (q >> 8);         // 0.110011 becomes 0.110011001100110011
  q = q + (q >> 16);        // Now have 32 bit's worth of product.
  return q >> 3;            // Doing this shift last of all
                            // gave better intermediate accuracy.
}
```

More on `http://www.hackersdelight.org/divcMore.pdf`

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 `demos/sixthcounting` for a demo where `float` gets this accidentally right and `double` gets it accidentally wrong.]

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 1.11...110 to 1.11...111, 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 $10^7 \leq 2^{24} \leq 10^8$.

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

[But you may only need 8 sig.figs if the decimal starts with 9.xxx

[See demos/printsigfig.]

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 still hold:

```
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
```

Overflow Exceptions

Overflow is the main potential source of exception.

Using floating point, overflow occurs when an exponent is too large to be stored.

Overflow exceptions most commonly arise in division and multiplication. Divide by zero is a special case of overflow.

But addition and subtraction can lead to overflow. When ?

Whether to raise an exception or to continue with a NaN?: If we return NaN it will persist under further manipulations and be visible in the output.

Underflow is normally ignored silently: whether the overall result is then poor is down to the quality of the programming.

[See demos/nan.]

Exceptions versus 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 results in scripting languages can return $0 + 1i$ if, unlike Java, they are untyped or dynamically typed 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!)

History: 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, available since Pentium and still used, includes a separate (better) instruction set which better enables compiler writers to generate fast (and IEEE-valid) floating-point code.

BEWARE: many x86 computers therefore have *two* floating point units! This means that a given program for x86 (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!)

Part 3

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. ± 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 $\texttt{asin}(x) \leq \texttt{asin}(y)$]

Unreported Errors in Floating Point Computations

Overflow is reported (exception or NaN).

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

- **quantisation errors** arising from 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.)
- **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).

(Elsewhere we define **truncation** errors and also speak of **underflow** and **cancellation/loss of significance**).

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.

Worst-Case 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)

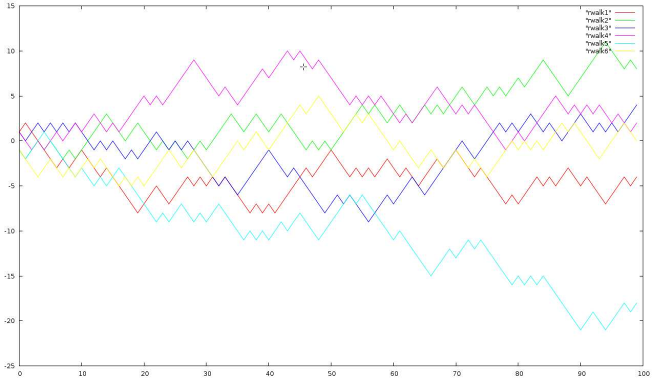
$$(x(1 \pm \eta_x)) * (y(1 \pm \eta_y)) = (x * y)(1 \pm (\eta_x + \eta_y) \pm \eta_x \eta_y)$$

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

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.

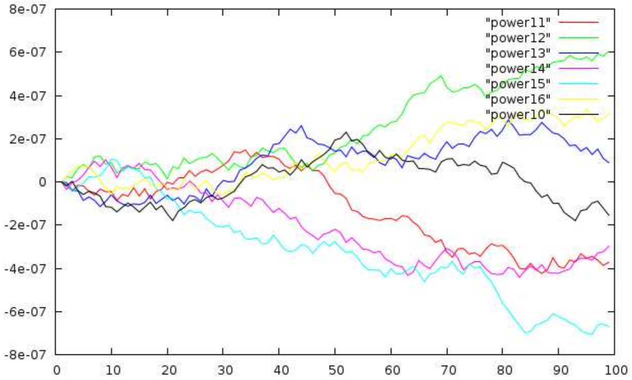
Random Walk Model for Error Accumulation

Summations 100 random values each of ± 1 (N=100, a=1).



Growth of relative error in p:

```
float p = 1.0, x = x̂ + 0.0;
for (i=0;i<100;i++) p=p * x;
```



If the steps are uncorrelated: expected sum is zero.

If the steps are uncorrelated: expected magnitude is $a\sqrt{N}$.

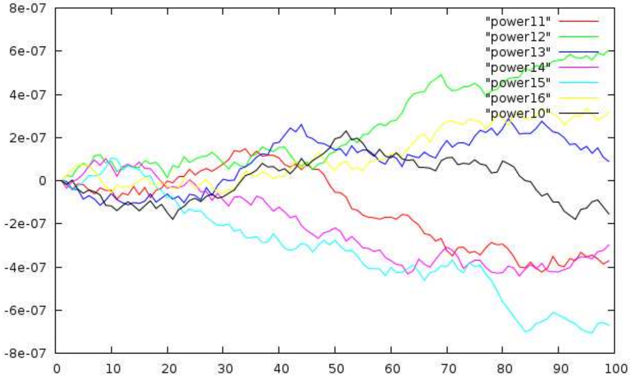
In the example, factor x was constant (possible correlated behaviour?).

Error Amplification is Typically More Important!

Growth of relative error in p:

float p = 1.0, x = $\hat{x}$ + 0;

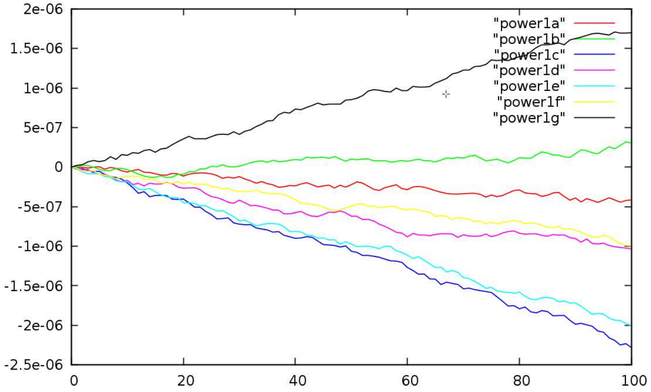
for (i=0;i<100;i++) p=p * x;



The same computation

when $x = \hat{x} \pm \epsilon$

($\hat{x}$ is not representable).



The first plots were for when x_k is exactly representable ($x_k = \hat{x_k}$).

For general $x(1 \pm \eta)$, final relative error is 100η [work it out yourself].

Simple worst-case analysis dominates random walk model.

Gradual loss of significance

Consider the program (c.f. the calculator example earlier)

```
double x_magic = 10.0/9.0;  // x = 1.111111111 ...
double x = x_magic;
for (i=0; i<30; i++)
{   printf("%e\n", x);
    x = (x-1.0) * 10.0;
}
```

[Look at demos/loss_sig output on the laptop]

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

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.

What happened: Worst-Case Analysis

Assume $x_0 = \hat{x} + \alpha$.

Then we repeatedly do $x_{n+1} := (x_n - 1.0) * 10.0$

Subtract step: x has abs error α and 1.0 is exact, so resultant has abs error α.

Multiply step:

1. Convert abs to rel: we know $x - 1 \simeq 0.111$ so $\eta = 1/0.111 \times \alpha \simeq 10\alpha$
2. 10.0 is exact, so rel error from multiplication unchanged: $\eta = 10\alpha$
3. Convert rel back to abs: value is approx 1.11 so $\epsilon = \eta = 10\alpha$

We accurately predict the factor of 10 growth in error.

Another starting value would overflow sooner, but the magic starting value hides the problem. (The iteration is metastable at x_magic.)

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"

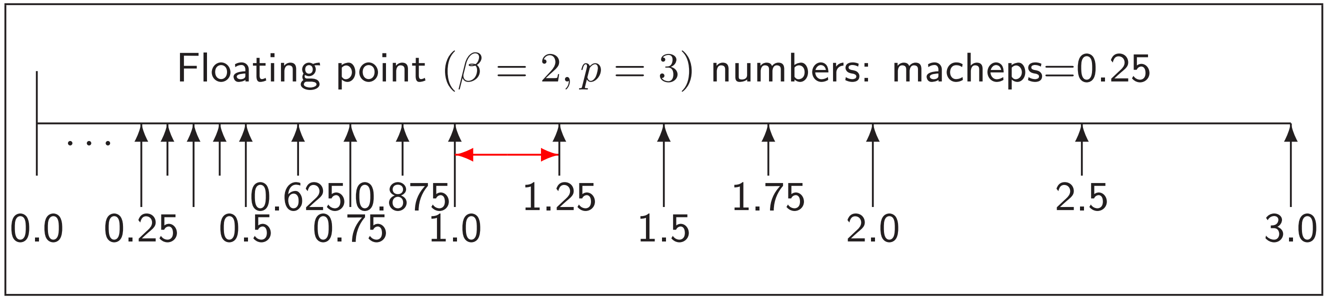
I.e. machine epsilon is 1 ulp for the representation of 1.0.

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

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



Floating point $(\beta = 2, p = 3)$ numbers: macheps=0.25
...
0.0
0.25
0.5
0.625
0.75
0.875
1.0
1.25
1.5
1.75
2.0
2.5
3.0

(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`

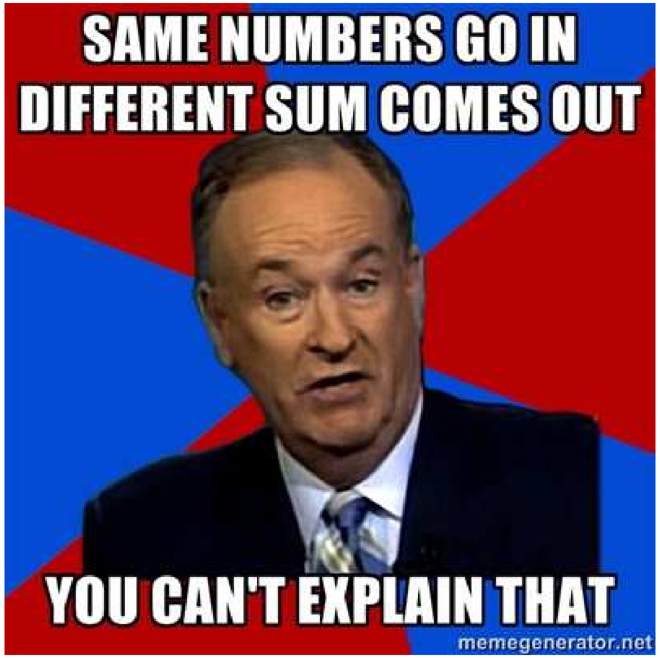
The answer given by `xcalc` earlier is totally bogus. Why?

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 $10^{40} \times \mathit{machine\_epsilon}$.) 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?

Part 4

Simple maths, simple programs


SAME NUMBERS GO IN
DIFFERENT SUM COMES OUT
YOU CAN'T EXPLAIN THAT
memegenerator.net

`http://www.pythian.com/blog/rounding-errors-in-mysql-with-decimal-type`

Hopefully the IEEE standard stopped that? Sorry...

Two Forms of Iteration

When calculating the sum to n we might use a loop:

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

```
s = 0;
for (int i=0; i<n; i++)  s +=  ....;
```

This is called a *map-reduce*: the work in the parallel steps is independent and results are only combined at the end under a commutative and associative operator. (But is FP addition commutative or associative?)

We have a totally different style of iteration when finding the root of an equation:

$$x_{n+1} = \frac{A/x_n + x_n}{2}$$

The way that errors build up or diminish is very different.

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 $b < 0$

[See demos/quadrat_float.]

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)

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

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 demos/SumFwdBack.]

Summing a Finite Series (2)

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

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

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.

For example: Compare

$$\begin{aligned}(14 + 14) + 250 &= 280 \quad \text{with} \\ (250 + 14) + 14 &= 270\end{aligned}$$

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) pin

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 - subsequent steps are likely to underflow less).

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 our running example, using a pair of single precision variables it yields:

- 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`

[demos/SumFwdBack on the course website.]

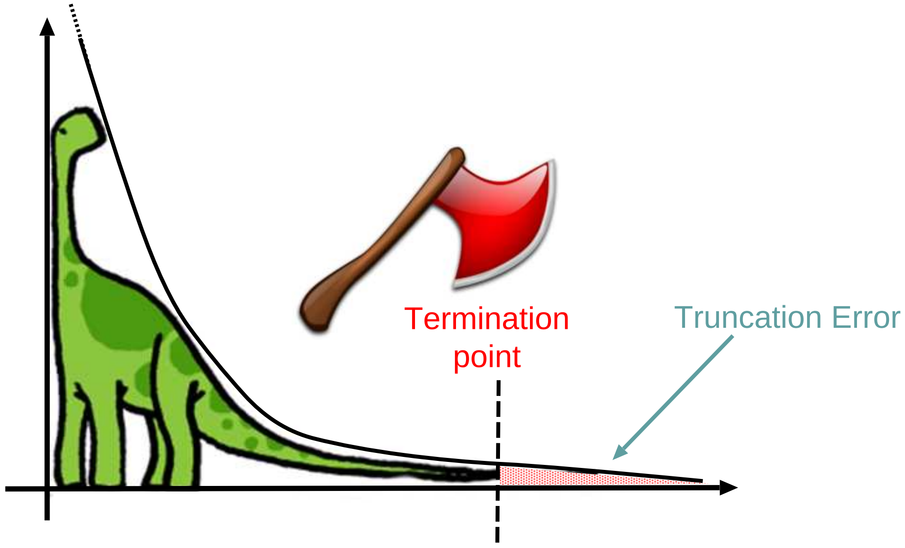
Infinitary/limiting computations

Rounding versus Truncation Error

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 and input quantisation error 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:

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

and it works out as approximately $hf''(x)/2$ [check it yourself],

i.e. it is roughly h given the assumption on f'' being around 1.

Illustration—differentiation (4)

For rounding error use Taylor again: assume $macheps$ error returned in evaluations of 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 `demos/DiffFloat` 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:

$$\frac{f(x+h) - f(x-h)}{2h}$$

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 $h = \sqrt[3]{macheps}$ is a good choice for h.

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 occur, e.g.

$$\int_0^1 f(x) \quad\approx\quad \frac{1}{n}\sum_{i=1}^{n} f(i/n) \qquad \text{where } h = \frac{1}{n}.$$

(Procedures for numerical integration are known as *quadrature* techniques.)

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).

Order of the Algorithm

The way in which *truncation error* is affected by reducing h (or increasing n) is called the *order* of the algorithm w.r.t. that parameter (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 reduces rounding error too).

Don't assume you can outguess the experts!

Root Finding

Commonly we want to find x such that $f(x) = 0$.

The Bisection Method: a form of *successive approximation*, is a simple and robust approach.

```
1. Choose initial values a, b such that sgn(f(a)) != sgn(f(b))
2. Find mid point c = (a+b)/2;
3. If |f(c)| < desired_accuracy then stop;
4. If sgn(f(c)) == sgn(f(a)) a=c; else b=c;
5. goto 2.
```

The absolute error is halved at each step so it has first-order convergence.

It is also known as binary chop and it clearly gives one bit per iteration.

(First-order convergence requires a number of steps proportional to the logarithm of the desired numerical precision.)

[The range find in float-to-ASCII used binary chop for the exponent.]

An Example of Convergence Iteration.

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	1.7320508075688772	1.1402e-01	
2	1.6528916502810695	3.4858e-02	3.0572e-01
3	1.6287699807772333	1.0736e-02	3.0800e-01
4	1.6213481984993949	3.3142e-03	3.0870e-01
5	1.6190578119694785	1.0238e-03	3.0892e-01
	. . .		
26	1.6180339887499147	1.9762e-14	3.0690e-01
27	1.6180339887499009	5.9952e-15	3.0337e-01
28	1.6180339887498967	1.7764e-15	2.9630e-01
29	1.6180339887498953	4.4409e-16	2.5000e-01
30	1.6180339887498949	0.0000e+00	0.0000e+00
31	1.6180339887498949	0.0000e+00	nan
32	1.6180339887498949	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:

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

I.E. first-order (or linear) convergence.

Golden ratio iteration (3)

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

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 example iteration this was $1/2\phi$.

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: $\epsilon_{n+1} \approx 3.236\epsilon_n$?

[See demos/Golden and demos/GoldenBad]

Golden ratio iteration (5)

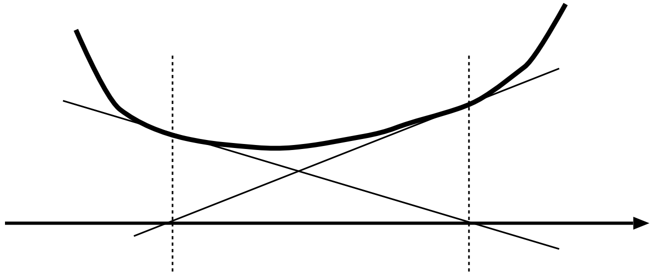
Yes! This is a bad iteration (magnifies errors).

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

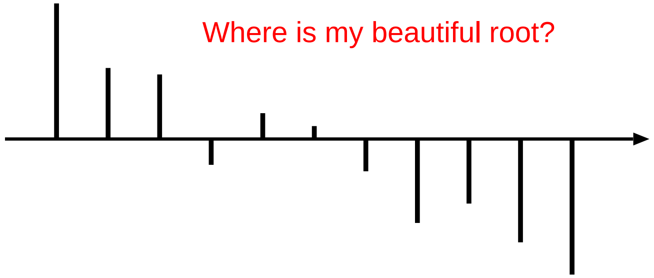
i	x	err	err/prev.err
1	1.61803398874989 51	2.2204e-16	
2	1.6180339887498958	8.8818e-16	4.0000e+00
3	1.6180339887498978	2.8866e-15	3.2500e+00
4	1.6180339887499045	9.5479e-15	3.3077e+00
5	1.6180339887499260	3.1086e-14	3.2558e+00
6	1.6180339887499957	1.0081e-13	3.2429e+00
7	1.6180339887502213	3.2641e-13	3.2379e+00
	. . .		
32	3.9828994989829472	2.3649e+00	3.8503e+00
33	14.8634884189986121	1.3245e+01	5.6009e+00
34	219.9232879817058688	2.1831e+02	1.6482e+01

Limit Cycles

An iteration may enter an infinite loop called a **Limit Cycle**.



1. This may be mathematically correct (e.g. Newton iteration of a pseudo root where the function just fails to kiss the x-axis).
2. It may arise from discrete (and non-monotone) floating point details.



Termination Condition Summary

When should we stop our iteration ?

1. After a pre-determined cycle count limit ?
2. When two iterations result in the same value?
3. When the iteration moves less than a given relative amount ?
4. When the error stops decreasing ? (When back substitution is cheap.)
5. When error is within a pre-determined tolerance?

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 or use an off-the-shelf package.

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 *second-order (or quadratic)* convergence.

Newton–Raphson (3)

Pros and Cons:

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

Quadratic (second-order) convergence requires a number of steps proportional to the logarithm of the desired number of digits.

Summing a Taylor series

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

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

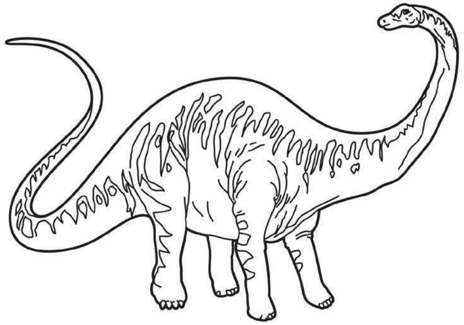
Mathematically, this is as nice as you can get—it unconditionally converges everywhere. However, computationally things are trickier.

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.



Thin at one end, getting ever so much fatter in the middle, and then thin at the other end!

[See `demos/SinSeries`.]

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 be 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.

Range Limit and Reduction: Exponential Function Example

Range limit:

x	$\exp(x)$	x	$\exp(x)$
-Inf		x $<$ -745.1	
+Inf		x $>$ 709.8	
NaN		otherwise	proceed

$$y = \exp(x) = \exp(k \ln 2 + r) = 2^k + \exp(r)$$

where $|r| <= 1/2 \ln 2$, so

$$\begin{aligned} k &= \lfloor 0.5 + x/(\ln 2) \rfloor \\ r &= x - k \ln 2 \end{aligned}$$

Example of range reduction: $\exp(12.34) = 2^{18} \times \exp(-0.1358)$

Then we might use Taylor $\exp(r) = 1 + r + r^2/2! + r^3/3! \ldots \; (0.347^{13}/13! = 1.6E - 16)$

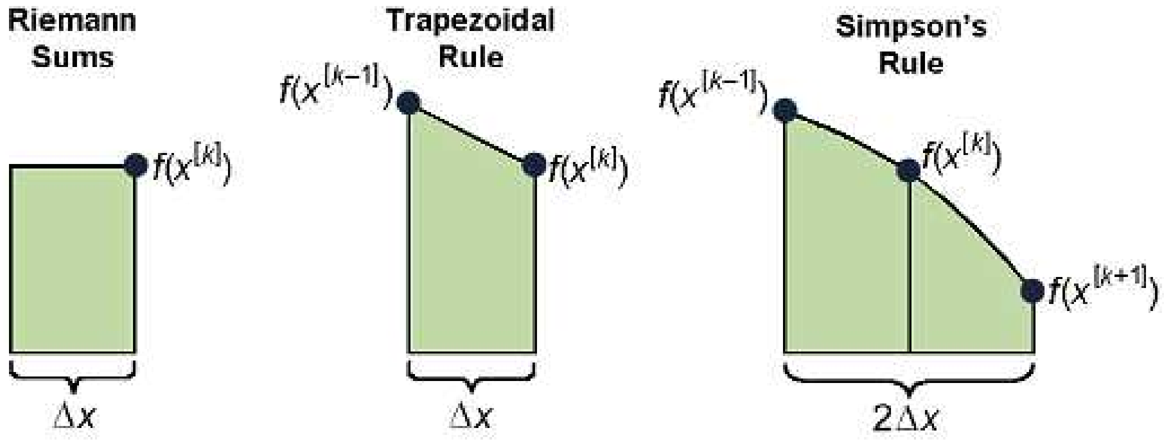
When $r >= 0 \Rightarrow$ All terms are positive $\Rightarrow$ Monotonic.

[Remez code really used on course web page.]

Commonly-used Quadrature Techniques (secondary school revision?)

Commonly we need a numerical approach to integrating a function between definite limits. In numerical analysis, methods for doing this are called **Quadrature** techniques.

Generally we fit some spline to the data and find the area under the spline.


Riemann Sums
$f(x^{[k]})$
Δx
Trapezoidal Rule
$f(x^{[k-1]})$
$f(x^{[k]})$
Δx
Simpson's Rule
$f(x^{[k-1]})$
$f(x^{[k]})$
$f(x^{[k+1]})$
$2\Delta x$

1. **Mid-point Rule** - puts a horizontal line at each ordinate to make rectangular strips,
2. **Trapezium Rule** - uses an appropriate gradiant straight line through each ordinate,
3. **Simpson's Rule** - fits a quadratic at each ordinate,
4. and clearly higher-order techniques are a logical extension...

Quadrature: How many strips to use?

$$\int_a^b f(x)\,dx \overset{\text{Simpson}}{\approx} \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right]$$

Rounding noise will be a random walk proportional to $\sqrt{n}$.

Truncation noise depends greatly on the nature of $f(x)$. It will be zero for quadratics and several classes of higher-order polynomial.

(A graph or two will be inserted here in the projection pack.)

[See `demos/simpson`.]

Lagrange Interpolation

Information transform theory tells us we can fit an n^{th} degree polynomial though the $n = k + 1$ distinct points: $(x_0, y_0),\ \ldots, (x_j, y_j),\ \ldots, (x_k, y_k)$.

Lagrange does this with the (obvious) linear combination

$$
\begin{aligned}
L(x) &= \sum_{j=0}^{k} y_j \ell_j(x) \\
\ell_j(x) &= \prod_{0\le m\le k}^{m\ne j} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)},
\end{aligned}
$$

The l_i are an orthogonal basis set: one being unity and the remainder zero at each datum.

Problem: Errors can be large at the edges. Fourier works well with a periodic sequence (that has no edges!).

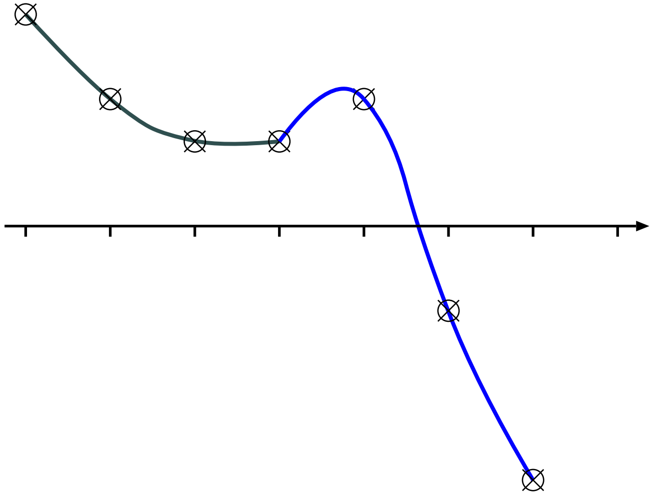
[See `demos/poly`.]

Splines and Knots

Splining: Fitting a function to a region of data such that it smoothly joins up with the next region.

Simpson's quadrature rule returns the area under a quadratic spline of the data.

If we start a new polynomial every k points we will generally have a discontinuity in the first derivative upwards.



Fitting cubics through four points with one point shared gives a continuous function with a potentially sharp corner!

If we overlap our splines we get smoother joins (Bezier Graphics next year).

Polynomial Basis Splines (B-Splines): May be lectured if time permits but non-examinable this year. This material is just an aside to demonstrate using a Chebyshev basis can be better than a naive approach.

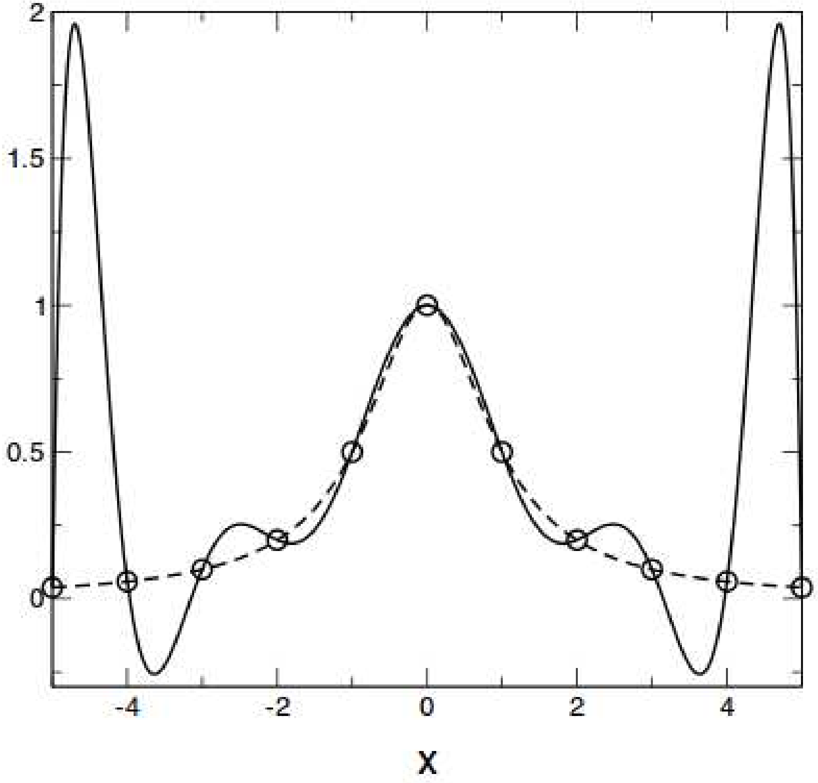
Poor Interpolation

The function $f(x) = 1/(1 + x^2)$ is notoriously poor under simplistic (linearly spaced (cardinal) Lagrange) interpolation.

If we fit an 10^{th}-order polynomial through the 11 points $\pm5, \pm4 ... 0$ we see large errors.

Dashed is the function.

Solid is our polynomial interpolation.



Better Interpolation

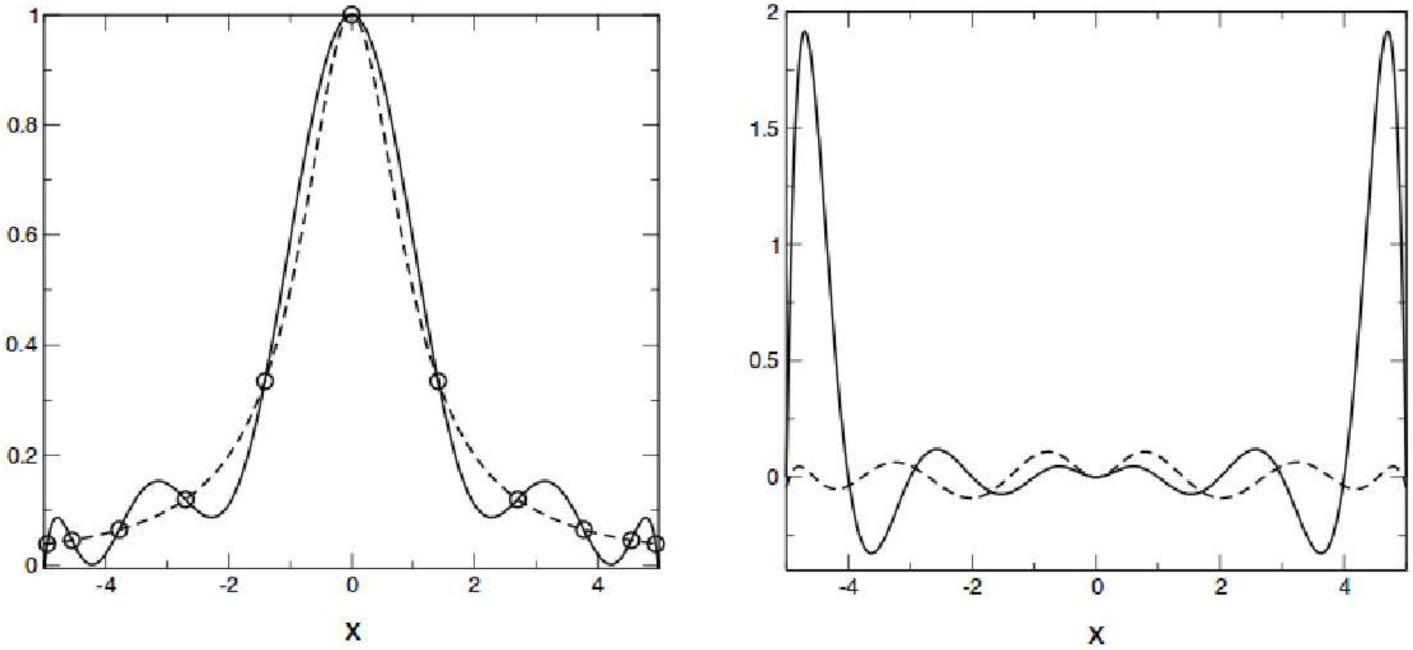


Figure 3.2. *Left: the function* $f(x) = 1/(1+x^2)$ *is plotted together with the interpolation polynomial for the* 11 *Chebyshev points Right: the interpolation errors for equispaced points and Chebyshev points is shown.*

Using Chebyshev knot spacing (linear cosine) we get a much better fit.

From "Numerical Methods for Special Functions" by Amparo Gil, Javier Segura, and Nico Temme.

Summing a Taylor series (5)

Using **Chebyshev Polynomials** we achieve *Power Series Economy*.

The total error in a Taylor expansion is re-distributed from the edges of the input range to throughout the input range.

Chebyshev orthogonal basis functions:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ T_d(x) &= 2xT_{d-1}(x) - T_{d-2}(x) \end{aligned}$$

Conversely, any monominal can be expressed:

$$\begin{aligned} 1 &= T_0 \\ x &= T_1 \\ x^2 &= (T_0 + T_2)/2 \\ x^3 &= (3T_1 + T_3)/4 \\ x^4 &= O(T_4)\ldots \end{aligned}$$

$$\forall n : -1 \leq x \leq +1 \Longrightarrow -1 \leq T_n(x) \leq +1$$

[See demos/chebyshev]

Summing a Taylor series (5b)

If we write our function as a linear sum of Chebyshev polynomials it turns out we can simply delete the last few terms for the same accuracy as delivered by Taylor.

$$f(x) \approx \sum_i a_i T_i(x)$$

(For those good at maths, but not examinable in this course: because the Chebyshev polynomials are an orthogonal basis set we can find their coefficients using the normal inner-product method as used, say, in Fourier transforms.)

We then simply collate on powers of x to get regular polynomial coefficients from the Chebyshev expanion.

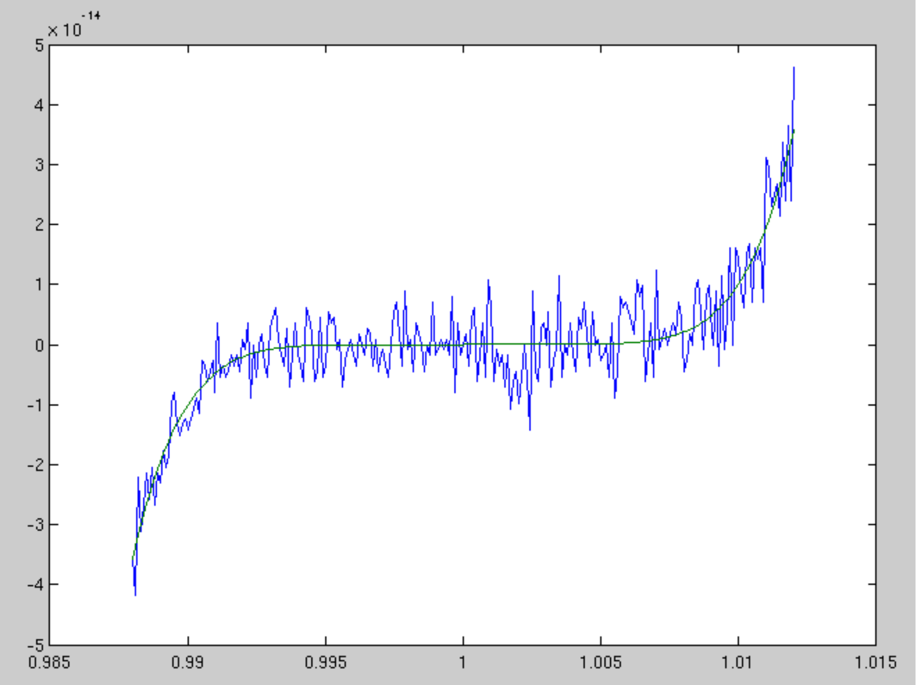
The new coefficients are actually just small adjustments of Taylor's values. Computation then proceeds identically.

The Chebyshev basis will be provided if needed for examinations.

Summing a Taylor series (6)

Run the Matlab/Octave file `demos/poly.m` from the course web-site to explore how careless factorisation of polynomials can cause noise.

$$\begin{aligned} y_0(x) &= (x-1)^7 \\ y_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \end{aligned}$$



Exercise: compare the roughness in the plot to your own estimate of the absolute error.

Real Example: Compute Taylor Series for Sine

```
fun sine x =
   let sq = - x * x

   let sine1 n sofar diff =
       let diff = (diff * sq / (n * (n+1))
       let x = diff + ans
       in if diff < margin then return x else sine1 (n+2) x diff

   in sine1 2 x x
```

$$sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - ...$$

But we cannot practically compute Chebyshev coefficients as we go. These must be precomputed and stored in a table.

[See demos/sin_series.]

How accurate do you want to be ? 1974 Sinclair Scientific Calculator

http://www.hackersdelight.org/

Total ROM used: a mere 320 words.

http://files.righto.com/calculator/sinclair_scientific_simulator.html

This page talks about how the 1974 Sinclair Scientific calculator was written, including recreating the source code in its entirety (and how logs, trig, and so on were coded and why). Lots of gems, e.g., the use of 0 and 5 to represent positive and negative sign (since 0+0=0 and 5+5=0). My favourite gem of all: *‘Scientific calculators usually provide constants such as e and but there was no space in the ROM for these constants. The Sinclair Scientific used the brilliant solution of printing the constants on the calculator’s case - the user could enter the constant manually if needed.’*

CORDIC (Volder's Algorithm)

CORDIC - COordinate Rotation DIgital Computer.

CORDIC became widely used in calculators and PCs of the 70's when the multiplications needed for Taylor's expansion were too slow.

We can find sine and cosine of Θ by rotating the unit vector $(1, 0)$.

$$\begin{pmatrix} \cos\Theta \\ \sin\Theta \end{pmatrix} = \begin{pmatrix} \cos\alpha_0 & -\sin\alpha_0 \\ \sin\alpha_0 & \cos\alpha_0 \end{pmatrix} \begin{pmatrix} \cos\alpha_1 & -\sin\alpha_1 \\ \sin\alpha_1 & \cos\alpha_1 \end{pmatrix} \cdots \begin{pmatrix} \cos\alpha_n & -\sin\alpha_n \\ \sin\alpha_n & \cos\alpha_n \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\Theta = \sum_i \alpha_i$$

Basis: successively subdivide the rotation into a composition of easy-to-multiply rotations until we reach desired precision.

[See demos/cordic]

CORDIC (Volder's Algorithm) (2)

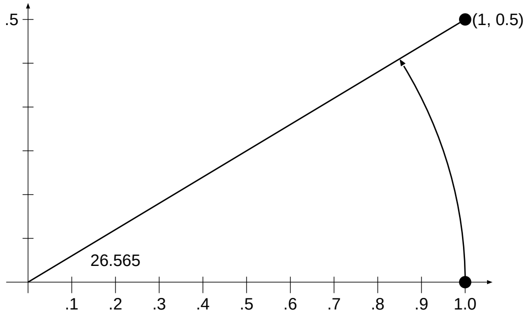
$$\cos\alpha \equiv \frac{1}{\sqrt{1+\tan^2\alpha}} \qquad R_i = \frac{1}{\sqrt{1+\tan^2\alpha_i}}\begin{pmatrix} 1 & -\tan\alpha_i \\ \tan\alpha_i & 1 \end{pmatrix}$$

$$\sin\alpha \equiv \frac{\tan\alpha}{\sqrt{1+\tan^2\alpha}}$$

Use a precomputed table containing decreasing values of α_i such that each rotation matrix contains only negative powers of two.

The array multiplications are then easy since all the multiplying can be done with bit shifts.

$$\begin{pmatrix} 1 & -0.5 \\ 0.5 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix}$$



The subdivisions are: $\tan 26.565° \approx 0.5, \tan 14.036° \approx 0.25, \ldots$

CORDIC (Volder's Algorithm) (3)

Handle the scalar coefficients using a proper multiply with lookup value from a second, precomputed table. $\texttt{scales}[\mathrm{n}] = \prod_{i=0}^{n} \frac{1}{\sqrt{1+\tan^2 \alpha_i}}$.

Or if using a fixed number of iterations, start off with a prescaled constant instead of 1.0.

```
void cordic(int theta)
{ // http://www.dcs.gla.ac.uk/~jhw/cordic/
  int x=607252935 /*prescaled constant*/, y=0;   // w.r.t denominator of 10^9
  for (int k=0; k<32; ++k)
  {
    int d = theta >= 0 ? 0 : -1;
    int tx = x - (((y>>k) ^ d) - d);
    int ty = y + (((x>>k) ^ d) - d);
    theta = theta - ((cordic_ctab[k] ^ d) - d);
    x = tx; y = ty;
  }
  print("Ans=(%i,%i)/10^9", x, y);
}
```

32 iterations will (almost?) ensure 10 significant figures.

Exercises: Given that $\tan^{-1}(x) \approx x - x^3/3$ explain why we need at least 30 and potentially 40 for this accuracy. Is it a good job that arctan $0.5 > 22.5$ or does this not matter? How would you use CORDIC for angles greater than 45 degrees?

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.

Exercise: The examples sheet asks about approximations to the arctan function.

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.

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

Notes for C and Java users

- Constants `1.1` are type `double` unless you ask `1.1f`.
- `float`s are implicitly converted to `double`s at various points (e.g. in C 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; }
```

`The strictfp keyword on Java classes and methods forces all intermediate resul`
`to be strict IEEE 754 values as well.`

may give different results.

What does this return?

```
float f=0.67;
if (f == 0.67) System.out.print("yes");
else System.out.print("no");
```

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*.

A summary slide. We have seen the random walks earlier now.

Part 6

Some nastier issues

Ill-Conditionedness and Condition Number

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. $a^2 + b^2 + c^2 + d^2$). The lines then are *nearly parallel*.

Ill-conditionedness (3)

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

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

Big numbers from nowhere! Small absolute errors in p or q will be greatly amplified in x and y.

Consider this geometrically: we are projecting 2-D to nearly 1-D: so getting back will be tricky.

Ill-conditionedness (4)

The previous slide was craftily constructed from Fibonacci numbers.

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

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,

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

As we go further down the Fibonacci sequence the ratio of gradients gets ever closer.

[Of course, filling a matrix with successive values from an iteration probably has no use apart from generating pathological examples. Any iteration that oscillates alternately around its root will serve.]

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).

Insight is available by form,ing or calculating 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.

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

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} !! [See `demos/bad_matrix_inverse`.]

Ill-conditionedness (8)

There's more theory around, but one definition is useful: the (relative) *condition number* : the $(\log_{10})$ ratio of relative error in the output to that of the input. When there are multiple inputs and outputs we normally quote the worst condition number from any input to any output.

A problem with a high condition number is said to be **ill-conditioned**.

Example 1: Multiplication by a precisely-represented, large constant gives a large first derivative but the relative error is unchanged. Although the relative error could be potentially large, it is unchanged in this operation, and hence the operation is well-conditioned.

Using the log form of the condition number gives the number of significant digits lost in the algorithm or process:

$$f_{(\text{with condition number } 5)}(1.23456789) = p.qrs$$

NB: There are various definitions of condition number. I have presented one that is useful for Computer Scientists.

Condition Number: Basic Illustrative Examples

1. **A large function** - f(x) $= 10^{22}$ – well behaved, Cond=$-\inf$
2. **A large derivative** - f(x) $= 10^{22}x$ – well behaved, Cond=0.0
3. **A spiking function** - f(x) $= \frac{1}{0.001+0.999(x-1000.0)}$ – Has a nasty pole?
4. **A cancelling function** - f(x) $= x - 1000.0$ – looks well behaved but ...

$$\begin{aligned} f(x(1+\eta)) &= x(1+\eta) - 1000.0 \\ \text{Cond} = \log_{10}\left(\left|\frac{\eta'}{\eta}\right|\right) &= \log_{10}\left(\frac{f(x(1+\eta)) - f(x)}{\eta.f(x)}\right) \\ &= \log_{10}\left(\frac{x.\eta}{\eta(x-1000.0)}\right) \end{aligned}$$

Consider x=1000.01, then Cond=$\log_{10}(10^5) = 5$.

$$\begin{aligned} f(1000.01) &= 0.01 \\ f(1000.011) &= 0.011 \text{ A 1 ppm domain perturbation became 10\% in range.} \end{aligned}$$

Note: This material worked through on the overhead in lectures...

Who has heard of L’Hôpital’s Rule ?

Where a function is poorly behaved we can often get a better expression from it by either simple rearrangment (like the quadratic formula earlier) or using L’Hôpital’s Rule or variants of it.

If

$$\lim_{x \to c} f(x) = \lim_{x \to c} g(x) = 0 \text{ or } \; \pm\infty,$$

and

$$\lim_{x \to c} \frac{f'(x)}{g'(x)} \;\; \text{exists, and} \;\; g'(x) \neq 0 \text{ for all } x \text{ in } I \text{ with } x \neq c,$$

then

$$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}$$

An overlap of programming and mathematics: you need to know a mathematical ‘trick’ to apply and then insert a condition in your program to selectively apply it.

E.g. consider plotting $\sin(x)/x$.

Not examinable this year.

Backwards Stability

Some algorithms are **backwards stable** meaning an inverse algorithm exists that can accurately regenerate the input from the output. Backwards stability generally implies well-conditionedness.

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.

Monte Carlo Technique to detect Ill-conditionedness

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

1. Take the original problem and solve.
2. Then take many variants of the problem, each perturbing the value of one or more parameters or input variables by a few ulps or a fraction of a 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

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).

Or just use a dynamically varying ΔT as discussed later (FDTD).

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) or large step size and then refining.

Simulated Annealing (non-examinable) is a standard technique where large jumps and random decisions are used initially but as the **temperature** control parameter is reduced the procedure becomes more conservative.

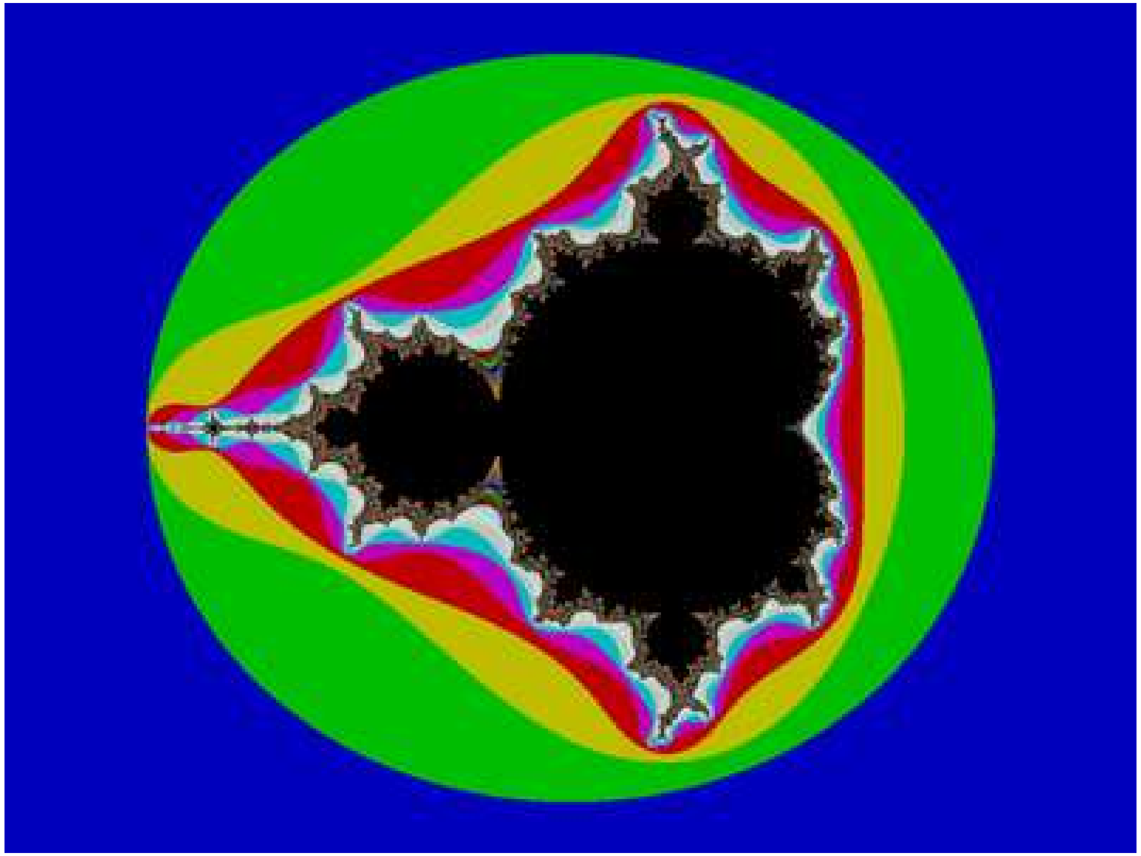
All of these are called "Adaptive Methods".

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 $x_{n+1} = rx_n(1 - x_n)$ 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 `demos/verhulst`.]

Mandelbrot set



Part 7

UNIVERSITY OF
CAMBRIDGE

Solving Systems of Simultaneous Equations

Solving Linear Simultaneous Equations

A system of linear simultaneous equations can be written in matrix form.

We wish to find $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$.

This can be found directly by finding the inverse

$$\mathbf{A^{-1}Ax} = \mathbf{A^{-1}b}$$

but finding the inverse of large matrices can fail or be unstable.

But note: having found the inverse we can rapidly solve multiple right-hand sides ($\mathbf{b}$ becomes a matrix not a vector).

Gaussian Elimination Method: we can freely add a multiple of any row to any other, so

1. do this to give an upper-triangular form,
2. then back substitute.

This is just a matrix phrasing of the school technique.

Example:

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 8\end{pmatrix}\begin{pmatrix}X_0\\ X_1\\ X_2\end{pmatrix} = \begin{pmatrix}1\\ 1\\ 0\end{pmatrix}$$

Add -4 times first row to the second:

$$\begin{pmatrix}1 & 2 & 3\\ 0 & -3 & -6\\ 7 & 8 & 8\end{pmatrix}\begin{pmatrix}X_0\\ X_1\\ X_2\end{pmatrix} = \begin{pmatrix}1\\ -3\\ 0\end{pmatrix}$$

Add -7 times first row to the third:

$$\begin{pmatrix}1 & 2 & 3\\ 0 & -3 & -6\\ 0 & -6 & -13\end{pmatrix}\begin{pmatrix}X_0\\ X_1\\ X_2\end{pmatrix} = \begin{pmatrix}1\\ -3\\ -7\end{pmatrix}$$

Take twice the second row from the third:

$$\begin{pmatrix}1 & 2 & 3\\ 0 & -3 & -6\\ 0 & 0 & -1\end{pmatrix}\begin{pmatrix}X_0\\ X_1\\ X_2\end{pmatrix} = \begin{pmatrix}1\\ -3\\ -1\end{pmatrix}$$

We now have upper-triangular form giving straightaway $X_2 = 1$.

Back substitute in second row $-3X_1 - 6 = -3 \leadsto X_1 = -1$.

Back substitute in first row $X_0 + 2 \times -1 + 3 = 1 \leadsto X_0 = 0$.

Complexity is: $O(n^3)$

Gaussian Elimination Minor Problems

The very first step is to multiply the top line by -$A_{0,1}/A_{0,0}$.

What if the **pivot element** $A_{0,0}$ is zero or very small?

A small pivot adds a lot of large numbers to the remaining rows: original data can be lost in underflow.

Note: Rows can be freely interchanged without altering the equations.

Hence: we are free to choose which remaining row to use for each outer loop step, so always choose the row with the largest leading value.

Selecting the best next row is **partial row pivoting**.

Various other quick processes can also be used before we start: scaling rows so all have similar magnitudes, or permuting columns. Column permutation requires we keep track of the new X_i variable ordering.

Permuting both rows and columns can lead to better solutions where all potential pivots otherwise found are small, but the search complexity is undesirable.

L/U and related Decomposition Methods

Note: Gaussian Elimination works for complex numbers and other fields as do all these methods.

If we have a number of right-hand-sides to solve is a good approach to first find the inverse matrix?

IE: To find x in $Ax = b$ we could use $x = A^{-1}b$.

In general, a better approach is to first triangle decompose $A = LU$, then

1. find y from $Ly = b$ using forwards substitution with the triangular form L,
2. then find x from $Ux = y$ using backwards substitution.

Complexity of a forwards or backwards substitution is quadratic.

[See demos/lu-decomposition.]

L/U Decomposition (2) (L/D/U is similar)

Find L and U such that $A = LU$ or find L, D and U such that $A = LDU$.

L/U details:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

Dolittle algorithm:

Do a normal Gaussian elimination on A to get U (ignoring any r.h.s to hand), but keeping track of the steps you would make on a r.h.s in an extra matrix. The extra matrix turns out to be L.

Write Gaussian step i on the r.h.s as a matrix multiply with G_i. This has close to identity form:

Then $L = \prod_i^{1..N} G_i$

The details are simple if you try an example.

You'll see you can simply form L in place starting from I.

$$G_i = \begin{pmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & -a_{r,c}/a_{i,i} & \ddots & & \\ & & \vdots & & \ddots & \\ 0 & & -a_{N,c}/a_{i,i} & & & 1 \end{pmatrix}.$$

Complexity: (an exercise).

Cholesky Transform for Symmetric +ve-definite Matrices

Decomposable, symmetric matrices commonly arise in real problems.

$$\mathbf{LL^T} = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix} = \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix} = \mathbf{A}$$

For any array size, the following Crout formulae directly give L:

$$\begin{aligned} L_{j,j} &= \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} \\ L_{i,j} &= \frac{1}{L_{j,j}}\left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{j,k}\right), \quad \text{for } i > j. \end{aligned}$$

Can now solve $Ax = b$ using:

1. find y from $Ly = b$ using forwards substitution with the triangular form L,
2. then find x from $L^T x = y$ using backwards substitution with L^T.

Complexity: $O(n^3)$. Stability: sqrt can fail: use L/D/U...

[See demos/cholesky-decomposition.]

What if I really do want the Matrix Inverse?

To find the inverse of $\mathbf{A}$, we commonly use our simultaneous equation solver to find the solution under the specific r.h.s. which is the identity matrix.

$$\mathbf{I} = \mathbf{A}\mathbf{A}^{-\mathbf{1}} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{23} \end{pmatrix} \begin{pmatrix} i_{00} & i_{01} & i_{02} \\ i_{10} & i_{11} & i_{12} \\ i_{20} & i_{21} & i_{23} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

In general, many specialised matrix techniques abound, especially for **sparse** and **banded** matrices.

What if my matrix is not Symmetric (not Hermitian)?

Magical ways have been worked out over the decades: we can use Cholesky to find $\mathbf{B^{-1}}$ even when $\mathbf{B}$ is not symmetric:

We first find the inverse of $(\mathbf{BB^T})$ which is always symmetric.

Then we correct the result using this identity:

$$\mathbf{B}^{-1} \equiv \mathbf{B}^{T}(\mathbf{BB^T})^{-1}$$

This slide is non-examinable.

What is a Hermitian Matrix ?

The Cholesky method requires a square, symmetric matrix for the real domain. In the complex domain it works for a Hermitian matrix.

A **Hermitian Matrix** is its own conjugate transpose: $A = \overline{A^T}$.

Hermitians are non-examinable on this course.

When to use which technique?

- Input is triangular ? - use in that form.
- Input rows OR cols can be permuted to be triangular ? - Use the permutatiion.
- Input rows AND cols can be permuted to be triangular ? - Expensive search unless sparse.
- Array is symmetric and +ve definite ? - Use Cholesky.
- None of above, but array at least is square ?
  - one r.h.s. - use Gaussian Elimination (GE),
  - multiple r.h.s. - use L/U decomposition.
- Array is tall (overspecified) ? - make a regression fit.
- Array is fat (underspecified) ? - optimise w.r.t. some metric function.

Simulated Annealing - Typical Pseudocode

```
temp := 200
ans := first_guess      // This is typically a long vector.
cost := cost_metric ans // We seek lowest-cost answer.

while (temp > 1)
{
   ans' := perturb_ans temp ans  // Magnitude of purt is proportional to temp
   cost' := cost_metric ans'

   accept := (cost' < cost) || rand(100..200) < temp;
   if (accept) (ans, cost, temp) := (ans', cost', temp * 0.99)
}
return ans;
```

Random perturbations at a scale proportional to the temperature are explored.
Backwards steps are occasionally accepted while above 100 degrees to escape local minima.
Below 100 we only accept positive progress (quenching).

Non-examinable in 2016/17 [See `demos/simulated_annealing`.]

Part 8

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 possess or 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)

Downsides:

- Will be slower (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.

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 *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.)

Example - MPFR Library

The GNU MPFR library from Inria is a C library for multiple-precision floating-point computations. It includes interval arithmetic. Most high-level languages have bindings for access without using C/C++.

```
#include <stdio.h>
#include <stdarg.h>
#include <mpfr.h>

int main(){
   mpfr_t x, result;
   mpfr_init2(x, 8192);
   mpfr_init2(result, 8192);
   mpfr_set_str(x, "1e40", 10, GMP_RNDN);
   int rounding = mpfr_sin(result, x, GMP_RNDN);
   mpfr_printf("Result: %.128Rf\n", result);
   return 0;
}
```

returns

```
Rounding result -1
Result:
-0.56963340095363632730803418157356872313292131914786845085382706375909300267931692421271067833769082619432451071422912793208209377
```

(Thanks to Timothy Goh for this example.)

Exact Real Arithmetic

There are a number of possible Exact Real Arithmetic packages, such as CRCalc, XR, IC Reals and iRRAM.

Consider using digit streams (lazy lists of digts) for infinite precision real arithmetic ?

Oh dear, it is impossible to print the first digit (0 or 1) of the sum $0.333333\cdots + 0.666666\cdots$ ...

Exact real arithmetic is beyond the scope of this course, but a great deal of work has been done on it.

[Demo of spigot presented in lectures.]

For those with interest: Arbitrary Precision Real Arithmetic: Design and Algorithms, Menissier-Morain, 1996: `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.8983` The first few 'striking examples' are rather stunning!

`http://www.chiark.greenend.org.uk/~sgtatham/spigot/spigot.html#examples`

Also, symbolic algebra: Maple, Mathematica and Arthur Norman's REDUCE.

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	$\underline{0.881836}$	$\underline{0.881836}$	$\underline{0.881836}$
5	$\underline{0.384327}$	$\underline{0.384327}$	$\underline{0.384327}$
10	$\underline{0.31303}4$	$\underline{0.313037}$	$\underline{0.313037}$
15	$\underline{0.0227}02$	$\underline{0.022736}$	$\underline{0.022736}$
20	$\underline{0.98}3813$	$\underline{0.982892}$	$\underline{0.982892}$
25	0.652837	$\underline{0.757549}$	$\underline{0.757549}$
30	0.934927	$\underline{0.481445}$	$\underline{0.481445}$
40	0.057696	$\underline{0.02400}8$	$\underline{0.024009}$
50	0.042174	$\underline{0.62}9402$	$\underline{0.625028}$
60	0.934518	0.757154	$\underline{0.315445}$

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

Pi to a trillion decimal places

In 2007 Yasumasa Kanada (Tokyo) exploited

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

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 break the world record for π digit calculation (1.2 trillion decimal places of accuracy).

September 2010 - N Sze used Yahoo's Hadoop cloud computing technology to find two-quadrillionth digit - the 2,000,000,000,000,000th digit.

Part 9

FDTD and Monte Carlo Simulation

FDTD=finite-difference, time-domain simulation.

Monte Carlo=Roll the dice, spin the wheel...

Simulation Jargon

Event-driven: A lot of computer-based simulation uses discrete events and a time-sorted pending event queue: these are used for systems where activity is locally concentrated and unevenly spaced, such as digital logic and queues. Topic is covered in later courses.

Numeric, finite difference: Iterates over fixed or adaptive time domain steps and inter-element differences are exchanged with neighbours at each step.

Monte Carlo: Either of the above simulators is run with many different random-seeded starting points and **ensemble** metrics are computed.

Ergodic: For systems where the time average is the ensemble average (i.e. those that do not get stuck in a rut): we can use one starting seed and average in the time domain (amortises start up transient).

Monte Carlo Example: Find Pi By Dart Throwing

```
double x = new_random(0.0, 1.0);
double y = new_random(0.0, 1.0);
if (x*x + y*y < 1.0) hit ++;
darts ++;
```

Do we expect the following convergence as $\mathrm{darts} \rightsquigarrow \mathrm{inf}$

$$\frac{\mathrm{hits}}{\mathrm{darts}} \rightsquigarrow \frac{\Pi}{4}$$

Exercise: What if we had used `new_random(-1.0, 1.0)` instead ?

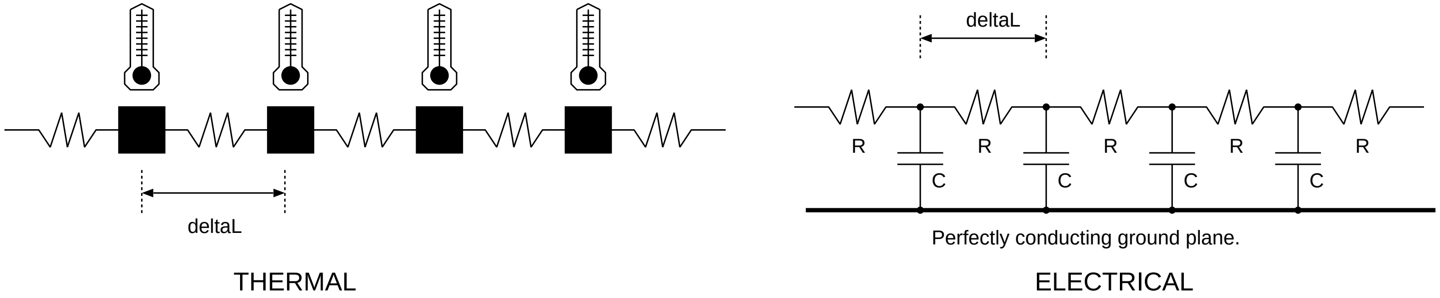
Exercise: What is the order of convergence for this dart throwing?

Aside: An interesting iteration to find Pi: `http://home.comcast.net/~davejanelle/mandel.html`

[See demos/dart-throwing.]

Finite-Difference Time-Domain Simulations (1)

Two physical manifestations of the the same 1-D example:


deltaL
THERMAL
deltaL
R
C
Perfectly conducting ground plane.
ELECTRICAL

The continuum of reality has been partitioned into finite-sized, elemental lumps.

Per element	Heat Conduction in a Rod	R/C Delay Line
$\rho A = dR/dx$	thermal resistance (J/s/$^\circ$C/m)	electrical resistance (Ω/m)
$\epsilon D = dC/dx$	thermal capacity ($^\circ$C/J/m)	electrical capacity (F/m)
Element state	Temperature ($^\circ$C)	Voltage (V)
Coupling	Heat flow (J/s)	Charge flow (Q/s=I)

In both systems, flow rate between state elements is directly proportional to their difference in state.

[See demos/RC-heatflow-1d.]

Newton's Laws of Mixing and Heat Conduction

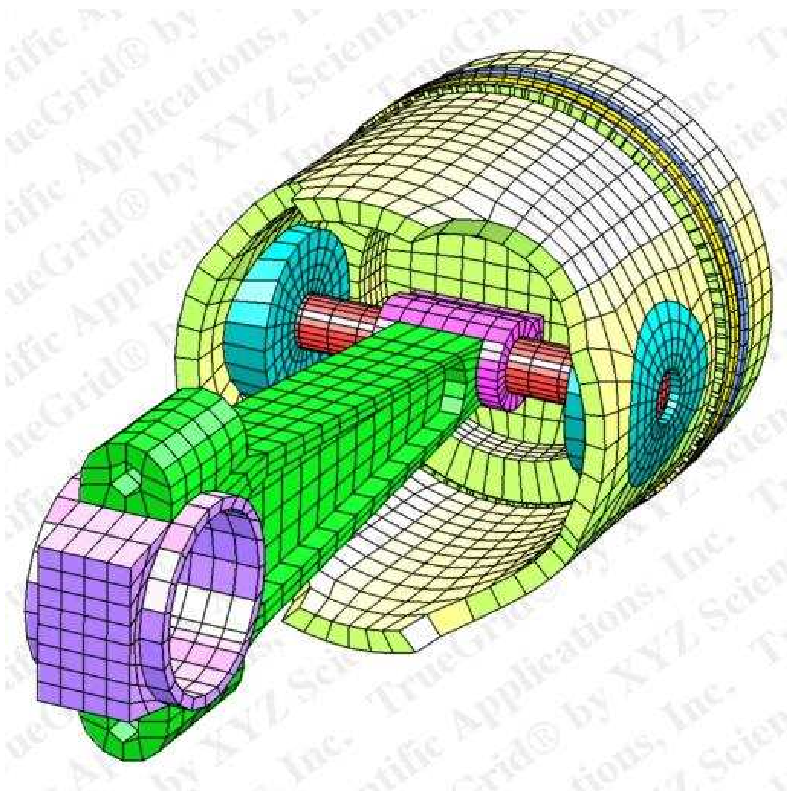
1. The temperature of an object is the heat energy within it divided by its heat capacity.
2. The rate of heat energy flow from a hotter to a cooler object is their temperature difference divided by their insulation resistance.
3. When a number of similar fluids is mixed the resultant temperature is the sum of their initial temperatures weighted by their proportions.

Finite-Difference Time-Domain Simulations (2)

More complex systems have a greater number of state variables per element and the elements interact in multiple dimensions.

For example, for this piston we might have temperature of each element and strain (deformation) in three dimensions.

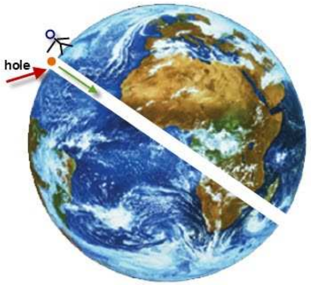
The **state vector** contains the variables whose values need to be saved from one time step to the next.



Exercise: Relate the 1-D heat or R/C electrical system to a third system: a water flow system where each element is a bucket with a given depth of water in it and the buckets are interconnected by narrow pipes between their bottoms.

Finite-Difference Time-Domain Simulations (3)

1. Red Ball Game: See `www.coolmath-games.com/0-red-ball-4`



2. Gravity Tunnel Through The Earth

   `http://hyperphysics.phy-astr.gsu.edu/hbase/mechanics/earthole.html`

3. Soyuz Rocket Simulation [See demos/soyuz.]

What ΔT step (and element size where relevant) do we need for a given accuracy?

Some of the above examples will be briefly lectured and included in the project-only copy of these notes.

Finite Differences Further Example - Instrument Physical Modelling

Physical modelling synthesis: Uses a FDTD simulation of the moving parts to create the sound waveform in real time.

Roland V Piano (2009): *Creating a digitally modelled piano is a fantastically difficult thing to do, but that hasn't deterred Roland... Introducing the V-Piano the worlds first hardware modelling piano!*

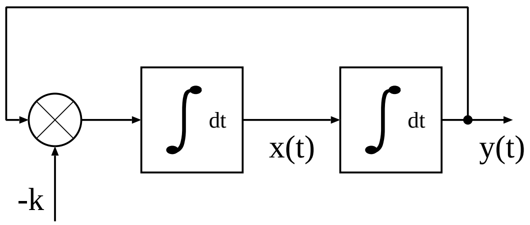
Example : A Violin Physical Model

1. Bow: need linear velocity and static and dynamic friction coefficients.
2. String: 2-D or 3-D sim?. Use linear elemental length 1cm? Need string's mass per unit length & elastic modulus.
3. Body: A 3-D resonant air chamber: 1cm cubes? Need topology and air's density and elastic modulus.

[See demos/RC-heatflow-1d/ViolinStringFDTD.]

Finite-Difference Time-Domain Simulations (4)

SHM: quadrature oscillator: generates sine and cosine waveforms.



Controlling equations:

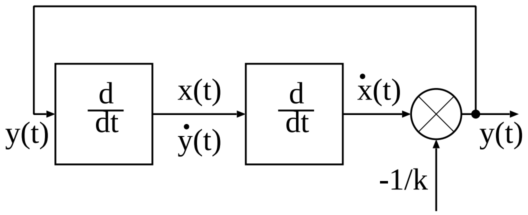
$$\begin{aligned} x &= \int -ky\,\mathrm{dt} \\ y &= \int x\,\mathrm{dt} \end{aligned}$$

For each time step:

$$\begin{aligned} x &:= x - ky\Delta t \\ y &:= y + x\Delta t \end{aligned}$$

[See demos/TwoPoleOscillators.]

Alternative implementation, using differentiators: is it numerically stable ?



Controlling equations:

$$\begin{aligned} \dot{x} &= -ky \\ \dot{y} &= x \end{aligned}$$

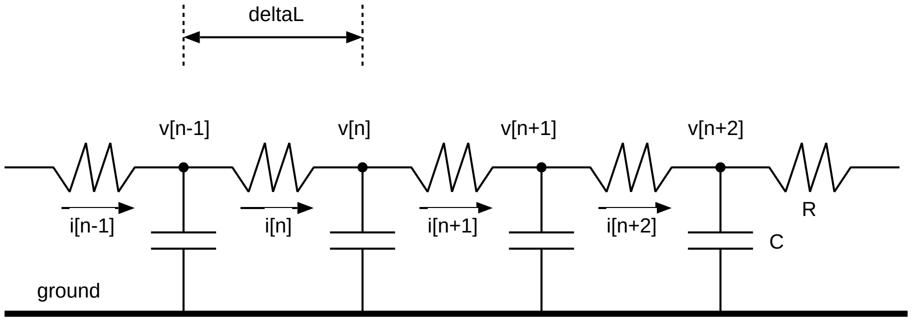
For each time step:

$$\begin{aligned} \dot{x} &:= (x - x_{\text{last}})/\Delta t \\ \dot{y} &:= (y - y_{\text{last}})/\Delta t \\ x &:= \dot{y} \qquad y := -\dot{x}/k \end{aligned}$$

... we cannot get further without some control theory (not lectured here!) ...

Iterating the Finite Differences

If we assume the current (heat flow) in each coupling element is constant during a time step



For each time step:

$$\begin{aligned} i[x] &:= (v[x+1]-v[x])/R \\ v[x] &:= v[x]+\Delta T(i[x+1]-i[x])/C \end{aligned}$$

But $i[x]$ need not be stored (it is not a state variable) giving instead:

$$v[x] \quad := \quad v[x]+\Delta T(v[x+1]-2v[x]+v[x-1])/RC$$

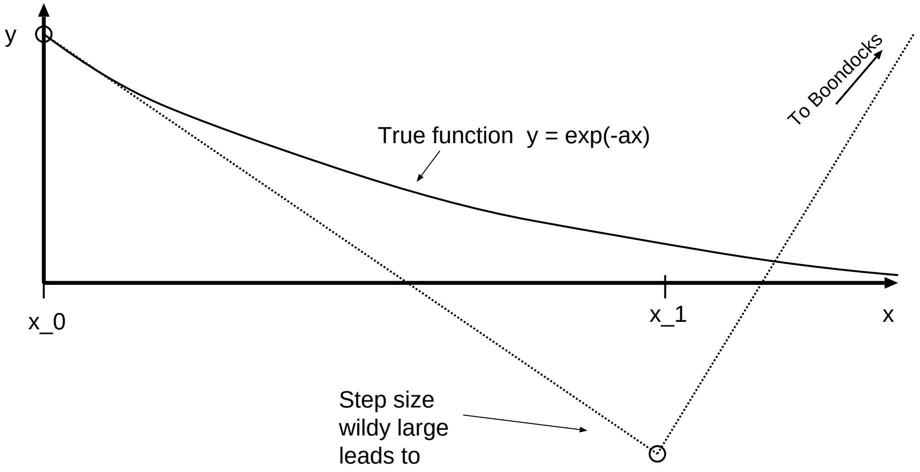
Modelling error source: in reality the current will decrease as the voltages (temperatures) of each element change (get closer?) during a time step.

Exercise/discussion point: Do they always get closer? Is this a systematic/predictable error and hence correctable?

Euler Method Stability

For many systems, forwards differences work well, provided we keep the step size sensible.

Simulation of $y = e^{-ax}$ fails if step size $h > 2/a$. (Proved in lecture).

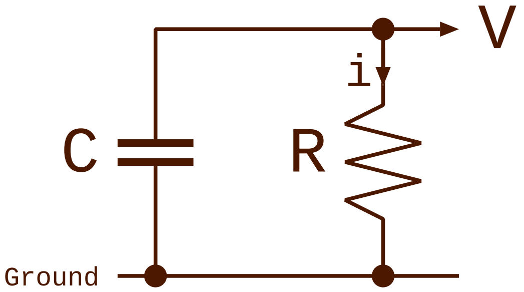

y
True function y = exp(-ax)
To Boondocks
x_0
x_1
x
Step size
wildy large
leads to

Systems that head for a unique equilibrium point can initially use a large step size without problem.

Forwards and Backwards Differences

The forward difference method (Euler's Method) uses the rate values at the end of one timestep as though constant in the next timestep.

End rates for the current step can (sometimes) be found from simultaneous equations.



We'll use a very simple example with one state variable: a capacitor discharging or heat dissipation from a blob.

Dynamic equations:

$$\begin{aligned} \mathrm{d}V/\mathrm{d}t &= -i/C \\ i &= V/R \end{aligned}$$

Dyns rewritten:

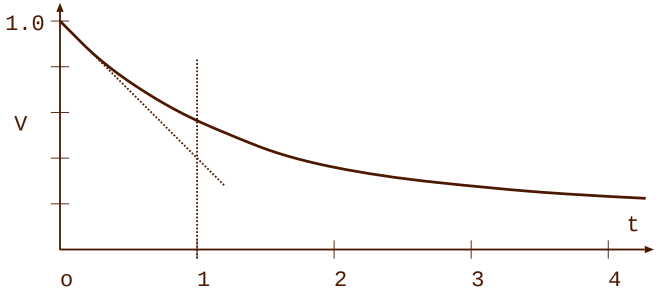
$$\begin{aligned} \mathrm{d}V/\mathrm{d}t &= -\alpha V \\ \alpha &= 1/CR \end{aligned}$$

Forward iteration:

$$\begin{aligned} V(n+1) &:= V(n) - \tau V(n) \\ \tau &= \Delta T/CR \end{aligned}$$

Forwards and Backwards Differences (2)

Numerical example, one time step, where $\alpha = 0.01$, $V(0) = 1.000000$:



True result given by

$$\hat{V}(t{+}1) = \hat{V}(t) \times e^{-t\alpha} = 0.990050$$

Forward stencil answer is $1 - \tau = 0.990000$.

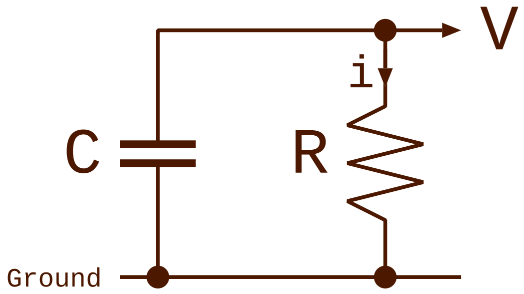
Using starting loss rate throughout timestep leads to an overcooling error of 50 ppm $\approx$ second order Taylor$(e^x) = \frac{1}{2}x^2$.

Halving ΔT, $\alpha = 0.005$, gives 0.990025 after two steps - half the error.

But is this a **systematic** error that will constructively reinforce?

Forwards and Backwards Differences (3)

We can potentially use the ending loss rate in whole time step:



The expressions for the ending values are found solving simultaneous equations. Here we have only one var so it is easy!

Reverse iteration:

$$\begin{aligned} V(n+1) &:= V(n) - \tau V(n+1) \\ &= V(n)/(1+\tau) \end{aligned}$$

Numeric result:

$$V(n+1) \quad := \quad 0.950099$$

So this under-decays...

Overall, combining equal amounts of the forward and backward stencils is a good approach: the **Crank-Nicolson method**.

Stability of Interacting Elements.

FDTD errors will tend to cancel out if either:

1. they alternate in polarity, or
2. they are part of a negative-feedback loop that leads to equilibrium.

Most systems that are simulated with FDTD have interactions between element pairs in both directions that are part of a stabilising, negative-feedback arrangement.

Even in the time domain, an exponential discharge using a forward stencil (using the loss rate at the start of a step for the whole step) means it ends up with less remaining than it should have, but then the rate of discharge will be correspondingly less in the following time step, stopping truncation error accumulation.

[See demos/forward-stencil-error-cancellation.]

Gear's Backwards-Forward Differentiation Method

Such time-domain patterns as these are known as **stencils**.

Numerical Integration of Stiff Ordinary Differential Equations Author: C W Gear

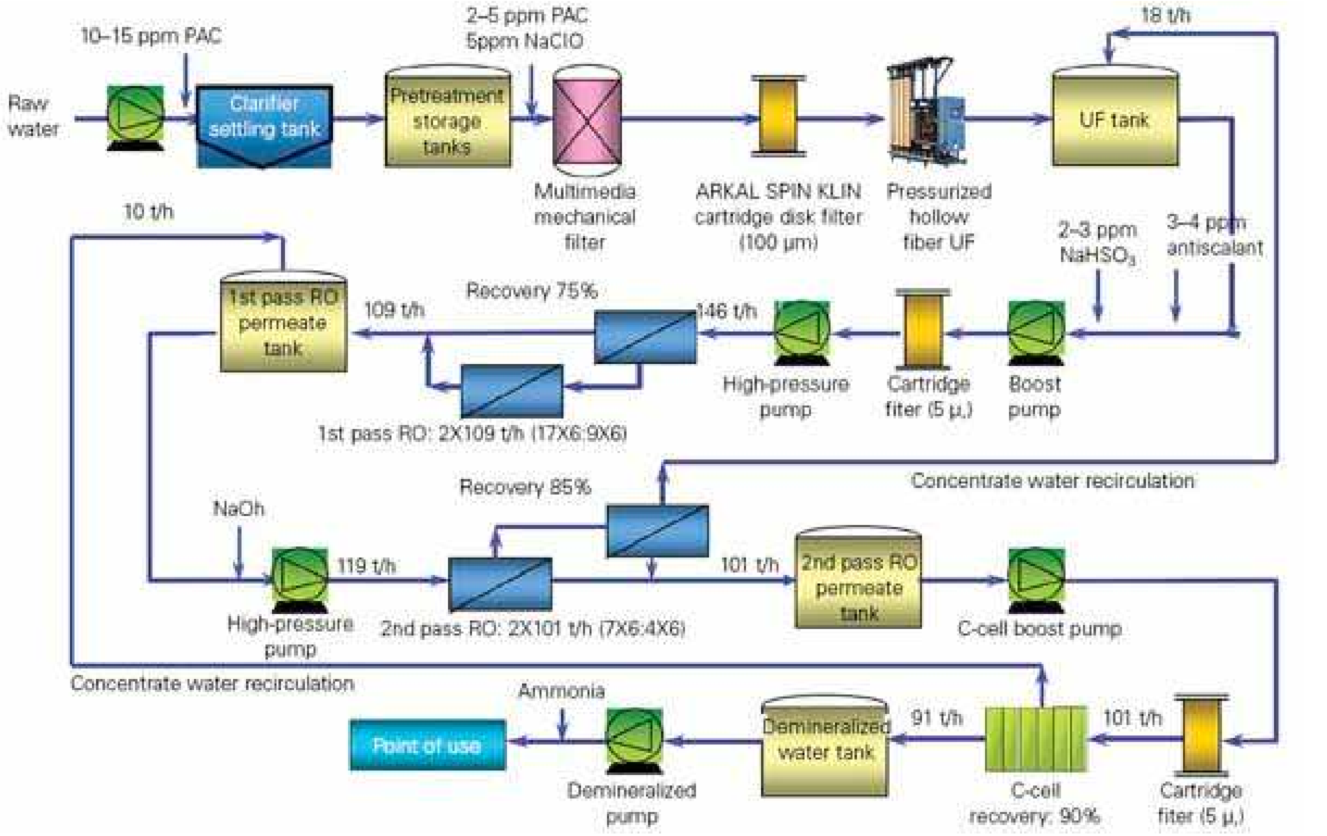
$$V(t+1) = V(t) \; + \; \frac{1}{3}(V(t) - V(t-1)) \; + \; \frac{2}{3}\Delta T \frac{dV}{dt}(t)$$

If we expect continuity in the derivatives we can get better forward predictions.

Equation sets that behave badly under forward differences are called **stiff.** The definition of 'stiff' is not known on Earth.

Non-examinable. Briefly lectured only.

Fluid Network (Circuit) Simulation

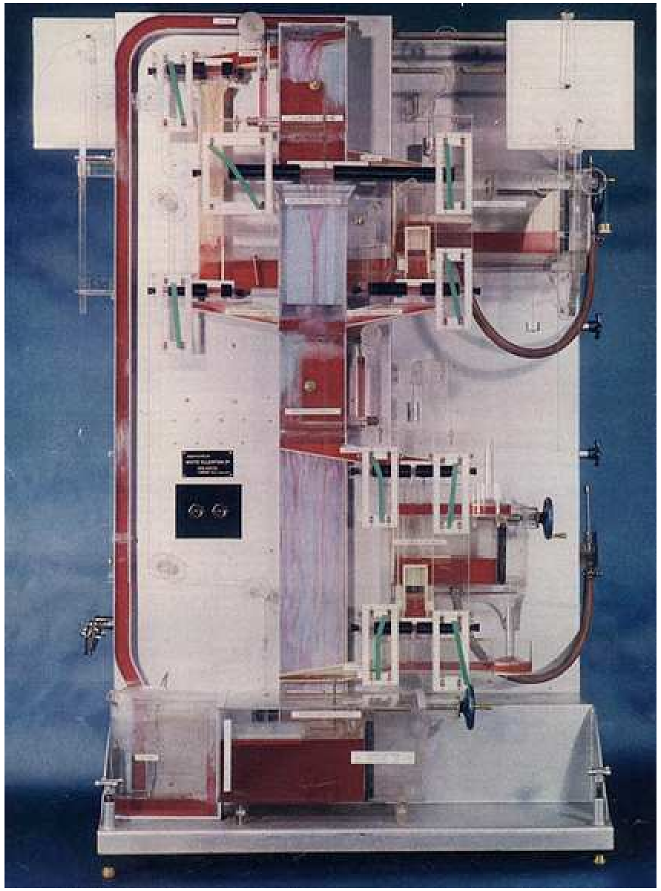

10–15 ppm PAC
Raw water
Clarifier settling tank
2–5 ppm PAC
5ppm NaClO
Pretreatment storage tanks
Multimedia mechanical filter
ARKAL SPIN KLIN cartridge disk filter (100 μm)
Pressurized hollow fiber UF
18 t/h
UF tank
2–3 ppm $NaHSO_3$
3–4 ppm antiscalant
10 t/h
1st pass RO permeate tank
109 t/h
Recovery 75%
146 t/h
High-pressure pump
Cartridge fiter (5 μ.)
Boost pump
1st pass RO: 2X109 t/h (17X6:9X6)
Recovery 85%
Concentrate water recirculation
NaOh
119 t/h
101 t/h
2nd pass RO permeate tank
High-pressure pump
2nd pass RO: 2X101 t/h (7X6:4X6)
C-cell boost pump
Concentrate water recirculation
Ammonia
Point of use
Demineralized pump
Demineralized water tank
91 t/h
C-cell recovery: 90%
101 t/h
Cartridge fiter (5 μ.)

Water Analogy of Circuits: The Moniac Economic Computer

http://en.wikipedia.org/wiki/MONIAC_Computer

MONIAC (Monetary National Income Analogue Computer) 1949 Bill Phillips of the LSE. There's one here in Cambridge at the Department of Economics.

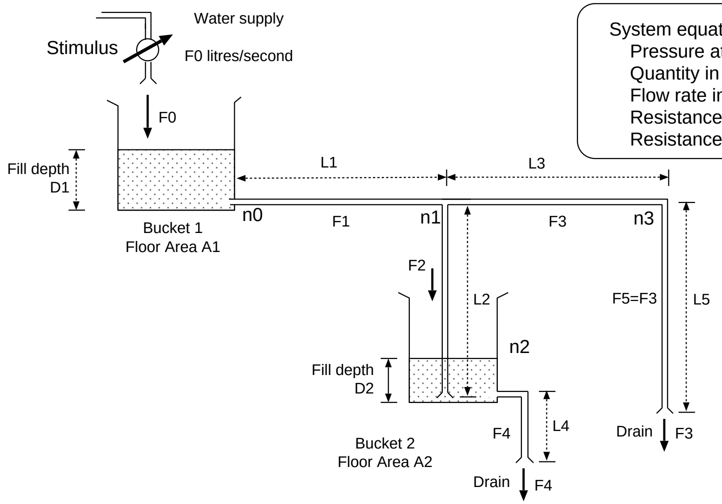
It models the national economic processes of the United Kingdom.



Exercise: Please see exercise sheet.

Water Flow Network

Can we write a FDTD simulation for a water circuit with non-linear components?


Water supply
Stimulus
F0 litres/second
F0
Fill depth
D1
Bucket 1
Floor Area A1
n0
L1
L3
F1
n1
F3
n3
F2
L2
F5=F3
L5
n2
Fill depth
D2
Bucket 2
Floor Area A2
F4
L4
Drain
F4
Drain
F3

System equations (linear terms only):
- Pressure at bottom of bucket = depth * rho_w
- Quantity in bucket = depth * area
- Flow rate in pipe = PressureDifference/Resistance
- Resistance of horizontal pipe = rho_h * length
- Resistance of vertical pipe = rho_v * length

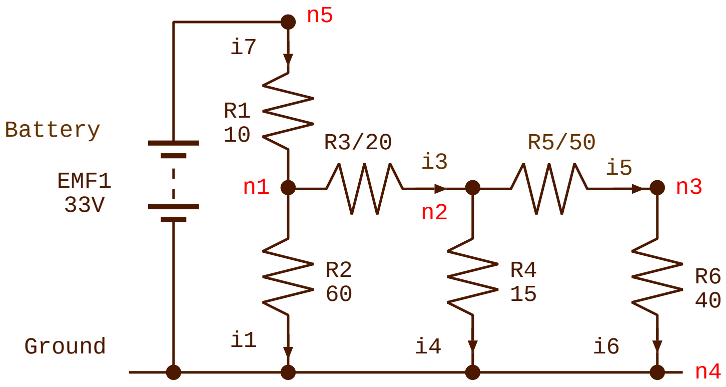
An analytical solution of the integral equations would have to assume pipes have linear flow properties: rate is proportional to potential/pressure difference (like Ohm's Law).

The vertical pipe is particularly non-linear in reality, having no effective resistance for low flow rates and the same resistance as a similar horizontal pipe for high flow rates. We'll learn how to model non-linear components...

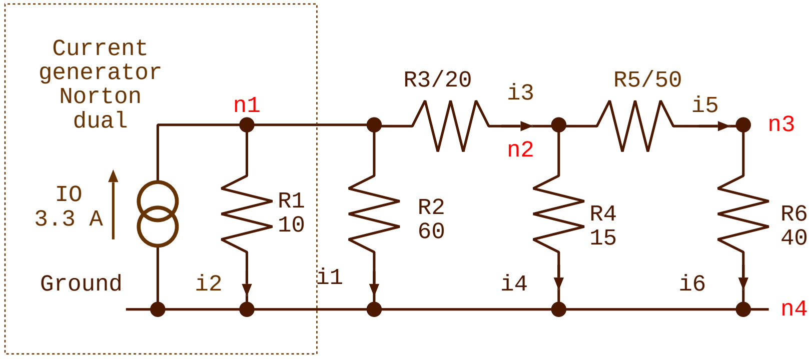
[See demos/water-network1.]

Circuit Simulation: Finding the Steady State (DC) Operating Point

Nodal Analysis: solve the set of flow equations to find node potentials.


n5
i7
R1
10
Battery
EMF1
33V
R3/20
i3
R5/50
i5
n1
n2
n3
R2
60
R4
15
R6
40
Ground
i1
i4
i6
n4

Electrical circuits generally have some voltage sources in them.

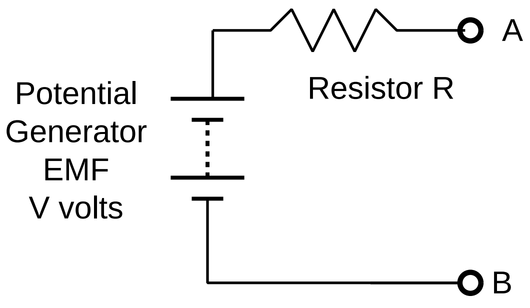

Current
generator
Norton
dual
n1
R3/20
i3
R5/50
i5
n3
n2
IO
3.3 A
R1
10
R2
60
R4
15
R6
40
Ground
i2
i1
i4
i6
n4

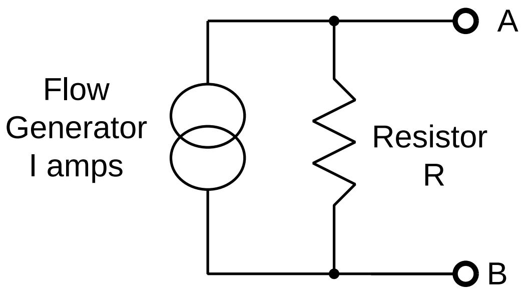
But our basic approach handles current sources only.

We will solve for the lower figure which is the Norton Dual (*non-examinable*) of the upper.

Norton and Thévinin Equivalents (or Duals).

A constant flow generator with a shunt resistance/conductance has exactly the same behaviour as an appropriate constant potential generator in series with the same resistance/conductance.


Potential
Generator
EMF
V volts
Resistor R
A
B


Flow
Generator
I amps
Resistor
R
A
B

Provided $V = IR$, both have the same current at and potential between their terminals when connected to:

1. infinite resistance (open circuit),
2. infinite conductance (short circuit),
3. and under all other load conditions.

Circuit Simulation: Finding the Steady State Operating Point (2)

Ohms law gives voltages from currents: I = V/R or I = GV.

The potential between two nodes is the current between them multiplied by the resistance between them. The interconnections between nodes are often sparse: rather than storing lots of nasty infinities we store conductance (G) values.

$$\begin{aligned} i_1 &= n_1/60 \\ i_2 &= n_1/10 \\ i_3 &= (n_1 - n_2)/20 \\ i_4 &= n_2/15 \\ i_5 &= (n_2 - n_3)/50 \\ i_6 &= n_3/40 \end{aligned}$$

Circuit Simulation: Finding the Steady State Operating Point (3)

Kirchoff's current law gives us an equation for each node: zero sum of currents.

$$\begin{aligned} -3.3 + i_2 + i_1 + i_3 &= 0 \\ -i_3 + i_4 + i_5 &= 0 \\ -i_5 + i_6 &= 0 \end{aligned}$$

Then solve G V = I Using Guassian Elimination (NB: A Symmetric +ve-def Matrix).

$$\begin{pmatrix} 1/10 + 1/60 + 1/20 & -1/20 & 0 \\ -1/20 & 1/20 + 1/15\underline{+1/50} & \underline{-1/50} \\ 0 & \underline{-1/50} & \underline{1/50} + 1/40 \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} = \begin{pmatrix} 3.3 \\ 0 \\ 0 \end{pmatrix}$$

Underlined is the template of the 50 ohm resisitor between nodes n_2 and n_3.

Component Templates/Patterns.

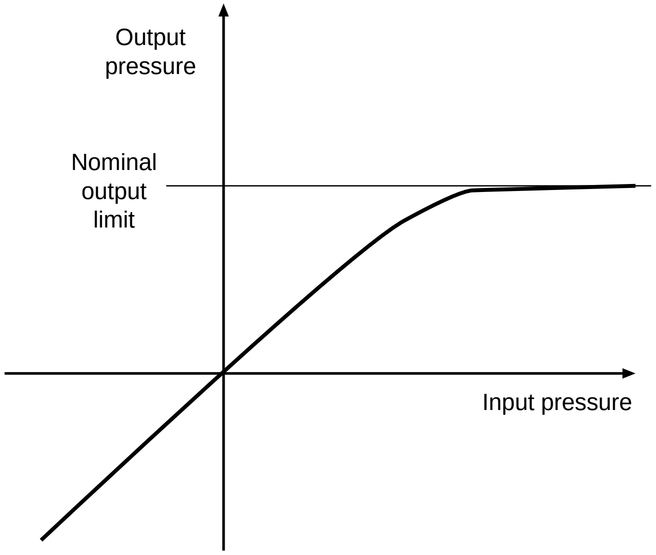
Conducting Channels: A conductance between a pair of nodes (x, y) appears four times in the conductance matrix. Twice positively on the leading diagonal at (x, x) and (y, y) and twice negatively at (x, y) and (y, x).
A conductance between a node x and any reference plane appears just on the leading diagonal at (x, x).
Adding positive values to the leading diagonal and negative values symmetrically elsewhere results in a +ve definite symmetric matrix. Hence Cholesky's method can be used.

Constant Flow Generators: A constant current generator appears on the right-hand side in one or two positions

Constant Potential Generators: In the basic approach, using only Kirchoff's flow law, the constant potential generators must be converted to current generators using a circuit modification. Modified nodal form (non-examinable) supports them directly. There the system is extended with voltage equations.

Circuit Simulation: Non-linear components.

Many components, including valves, diodes, vertical pipes, have strongly non-linear behaviour.



The resistance increases drastically above the target output pressure.

A Pressure Regulator Valve

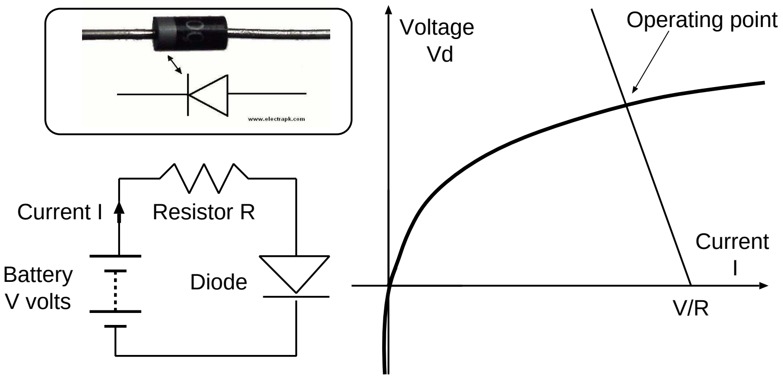
Note: electrical circuits will not be asked for in Tripos exams. This diode example, for instance, can be as well understood using a water flow analogy and a wier with non-linear height/flow behaviour. Non-linear links set the max in min-cut, max-flow analysis.

Circuit Simulation: Non-linear components (2).

In electrical circuits, the semiconductor diode is the archetypal non-linear component.

The operating point of this battery/resistor/diode circuit is given by the non-linear simultaneous equation pair:

$$\begin{aligned} I_d &= I_{\mathrm{S}}\left(e^{V_d/(nV_{\mathrm{T}})} - 1\right) && \text{(Shockley ideal diode equation)} \\ V_d &= V_{\mathrm{battery}} - I_d R && \text{(Composed battery and resistor equations)} \end{aligned}$$

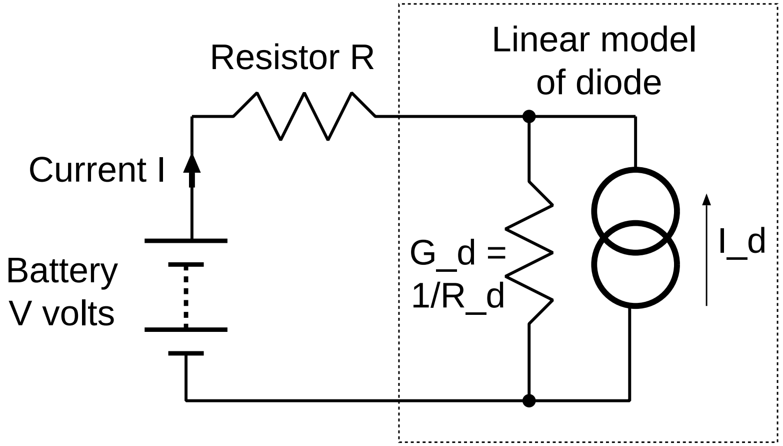

www.electrapk.com
Current I
Resistor R
Battery
V volts
Diode
Voltage
Vd
Operating point
Current
I
V/R

The diode's resistance changes according to its voltage! But a well-behaved derivative is readily available.

A non-linear component's tangent does not pass through the origin. An offsetting phantom generator must be added.

Circuit Simulation: Non-linear components (3)

Linearise the circuit: nominally replace the nonlinear components with linearised model at the operating point. In the electronic example, the non-linear diode changes to a linear current generator and conductance pair.



G_d and I_d are both functions of V_d.

$$\begin{aligned} G_d &= dI/dV = \frac{I_S}{nV_T} e^{V_d/(nV_T)} \\ I_d &= V_d G_d \end{aligned}$$

When we solve the circuit equations for a starting (G_d, I_d) pair we get a new V_d. We must iterate.

Owing to the differentiatable, analytic form of the semiconductor equations, Newton-Raphson iteration can be used.

Typical stopping condition: when relative voltage changes less than 10^{-4}.

Electrical details not examinable but generic linearisation technique is.

[See demos/DiodeExample.]

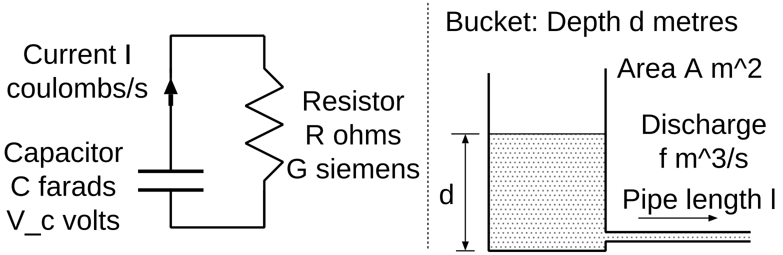
Circuit Simulation: Time-varying components

Charge leaves a capacitor at a rate proportional to the leakage conductance.

Water flows down our pipes at a rate proportional to pressure difference.

Both systems discharge at a rate proportional to the amount remaining.

$$\begin{aligned} \mathrm{d}V_c/\mathrm{dt} &= -I/C \\ I &= V_c G \end{aligned}$$


Current I
coulombs/s
Capacitor
C farads
V_c volts
Resistor
R ohms
G siemens
Bucket: Depth d metres
Area A m^2
Discharge
f m^3/s
Pipe length l
d

$$\begin{aligned} \mathrm{dd}/\mathrm{dt} &= -f/A \\ f &= \rho dg/kl \end{aligned}$$

At any timepoint, the capacitor/bucket has a fill level and a flow rate: and hence a ‘conductance’ ratio between them.

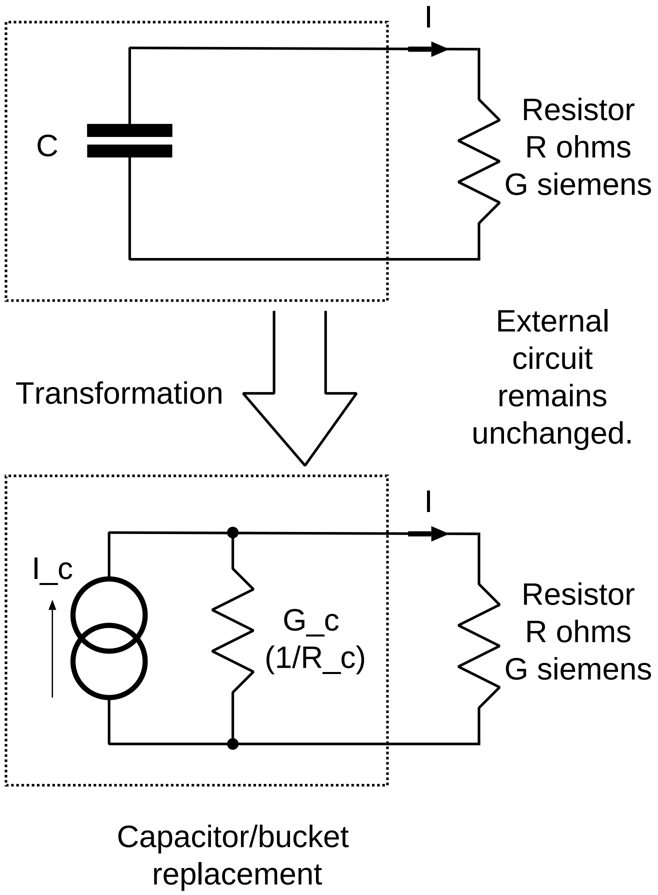
Perform time-domain simulation with a suitable ΔT.

(Unlike the nasty diode, a linear time-varying component like a capacitor or water tank has a direct analytical linear replacment so we do not need to iterate within the timestep etc..)

Non-examinable footnote: Electrical inductors are the same as capacitors, but with current and voltage interchanged. There is no ready equivalent for inductors in the water or heat analogies.

Circuit Simulation: Time-varying components (2)

Now we need to linearise the time-varying components using a snapshot: replace the bucket (capacitor) with a resistor and constant flow (current) generator.


I
C
Resistor
R ohms
G siemens
External
circuit
remains
unchanged.
Transformation
I
I_c
G_c
(1/R_c)
Resistor
R ohms
G siemens
Capacitor/bucket
replacement

$$\frac{\mathrm{d}V}{\mathrm{d}I} = \frac{\mathrm{d}(\Delta V)}{\mathrm{d}I} = \frac{\mathrm{d}(I\Delta T/C)}{\mathrm{d}I} = \frac{\Delta T}{C}$$

$$G_c = \frac{\mathrm{d}I}{\mathrm{d}V} = \frac{C}{\Delta T}$$

$$I_c = V_c G_c$$

We need one FDTD state variable.
The voltage (bucket fill depth):

$$V_c^{(t+\Delta T)} = V_c^{(t)} - \frac{I\Delta T}{C}$$

[See demos/TankExample.]

Circuit Simulation: Adaptive Timestep

Overall we have a pair of nested loops:

1. Iterate in the current timestep until convergence.
2. Extrapolate forwards in the time domain using preferred stencil (e.g. simple forward stencil) (a source of simulation error).

So, we need careful (adaptive) choice with ΔT:

1. too large: errors accumulate undesirably,
2. too small: slow simulation.

A larger timestep will mean more iterations within a timestep since it starts with the values from the previous timestep.

Smoother plots arise if we spread out this work in the time domain (trade convergence iterations for timesteps).

Iteration count adaptive timestep method:

```
if $N_{it} > 2N_{\max}$ { $\Delta T * = 0.5$; revert_timestep(); }
else if $N_{it} > N_{\max}$ { $\Delta T * = 0.9$; };
else if $N_{\mathrm{it}} < N_{\min}$ { $\Delta T * = 1.1$; }
```

The End.

Please do the online exercises and study the tiny example programs in the `demos` folder.

Thankyou and goodbye for now ...

©DJG April 2012-2018.