Numerical Methods
(slides 1–191)

An eleven-lecture course

Dr. D J Greaves
Computer Laboratory, University of Cambridge
`http://www.cl.cam.ac.uk/teaching/current/NumMethods`

Easter 2013–14 (R2g)

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!

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.

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

- Patriot missile interceptor fails to intercept due to 0.1 second being the 'recurring decimal' $0.00\,1100\,1100\ldots_2$ in binary (1991)

- Ariane 5 \$500M firework display caused by overflow in converting 64-bit floating-point number to 16-bit integer (1996)

- The Sleipner A offshore oil platform sank . . . post-accident investigation traced the error to inaccurate finite element approximation of the linear elastic model of the tricell (using the popular finite element program NASTRAN). The shear stresses were underestimated by 47% . . .

Learn from the mistakes of the past . . .

Castle Hill Problem

Certain types of curry lead to problems afterwards.

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.

Introduction/reminding you what you already know

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 audio). Note $1111.1111_2 = 15.9375_{10}$.

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

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 $1 \le a < 10$. 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.

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.

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

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

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)

Scientific form numbers (note unequal gaps)
log scale shows this clearly
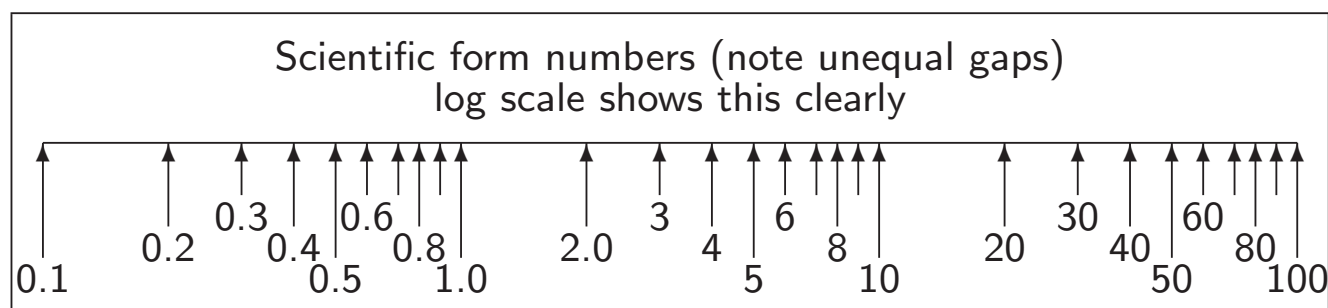
You might prefer to say $sig.figs.(4.56) = -\log_{10} 0.01/4.56$.

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

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

Get your calculator out!

Calculators are 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)

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

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

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:

$$\sin 1e40 = 0.3415751$$

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

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

- fix a size of mantissa (sig.figs.)

- fix a size for exponent (exponent range)

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

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

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

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

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

Now let's turn to binary representations and keep $\beta = 2$. This is as used on all contemporary general purpose machines.

Note: The IBM/360 series of mainframes used $\beta = 16$ which gave some entertaining but obsolete problems.

Decimal or Binary?

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

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

Most calculators work in decimal, $\beta = 10$, 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 easy: just read the value off the digit wheels in each register!
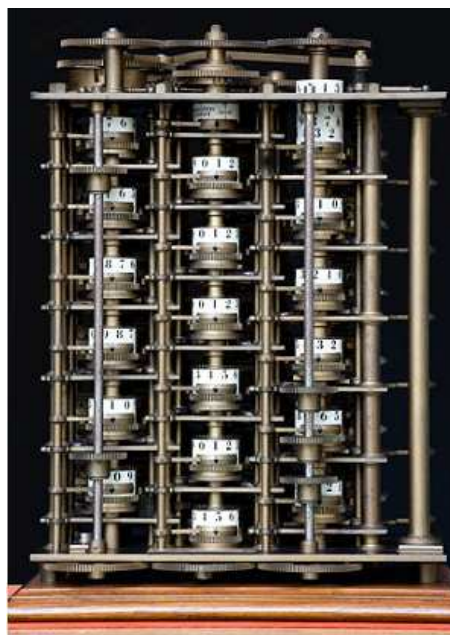
Exercise: IBM System 360 used (uses!) $\beta = 16$. Was (is) this a good idea?

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.

Exercises: Multidigit (long) multiplication is linear or quadratic in word length?

Give the code for base/radix 4 multiplication without Booth's optimisation.

Long Multiplication Algorithm - Broadside 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
```

Multiplying or dividing by powers of 2 uses shifts which are 'free' in binary hardware.

Booth's Algorithm

```
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.
```

Booth's algorithm avoids multiplying by 3 using a negative carry concept.

Overall one adder/subtractor is needed while working in base 4.

Basic Long Multiplication - from the Exercise Sheet

```java
final static int base = 10;
static int [] xd, yd, rd;


static void naive_method()
{
    for (int x=0; x<xd.length; x++)
        for (int y=0; y<yd.length; y++)
            {
                int b = x+y;
                int s = xd[x] * yd[y];
                //if (s> 0) System.out.println("x=" + x + " y=" + y + " s=" + s);
                while (s > 0)
                    {
                        s += rd[b];
                        rd[b++] = s % base;
                        s = s / base;
                    }
            }
    // Do we need to check most-significant result bit is less than base?
    }
```

Vehdic Multiplication - from the Exercise Sheet

```
static void vehdic_method()
{
    int b, c_in = 0;
    for (b=0; b<xd.length+yd.length; b++)
        {
            int s = c_in;
            for (int x=0; x<=b; x++)
                {
                    int y = b-x;
                    if (x >= xd.length || y >= yd.length) continue;
                    s += xd[x] * yd[y];
                }
            rd[b] = s % base;
            c_in = s / base;
        }
}
```

```
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 doesnt *)
                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
```

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?

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.

```
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.

```
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.

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.

The only important part is the central iteration (successive approximation) that scans one of these two lookup tables:

```
const int N_BIN_POWERS_OF_TEN = 6; // Range: 10**(+/- 63)
const long double bin_exponents_table[N_BIN_POWERS_OF_TEN] =
        { 1.0e1, 1.0e2, 1.0e4, 1.0e8, 1.0e16, 1.0e32 };
const long double bin_fracs_table[N_BIN_POWERS_OF_TEN] =
        { 1.0e-1, 1.0e-2, 1.0e-4, 1.0e-8, 1.0e-16, 1.0e-32 };
```

and the correction: `exponent += precision;`

Exercise: is the range $10^{\pm 63}$ sufficient?   *This is in C: There is an ML version online.*

```c
void doubleToStr(bool scientific_form,
                 double d0,
                 int precision)
{
  if (d0 < 0.0)
    { d0 = -d0;
      putchar('-');
    }
  union tobits
  { // Abuse union safety to get bits.
    double df;
    unsigned long long int di;
  } to;
  to.df = d0;
  int in_exp = (to.di >> 52) & 0x7FF;
  int mant = to.di & ((1llu<<52)-1));
  if (in_exp == 0x7FF)
    { if (mant == 0)
        { putchar('I');
          putchar('n');
          putchar('f');
        }
      else
        { putchar('N');
          putchar('a');
          putchar('N');
        }
      return;
    }
  // Render all denormals as 0.0
  if (in_exp == 0)
    { putchar('0');
      putchar('.');
      putchar('0');
      return;
    }

int exponent = 0;
unsigned int i,
bitpos = 1U<<(N_BIN_POWERS_OF_TEN-1);
int lower_bound = tens_table[precision];
int upper_bound = tens_table[precision+1];
if (d0 < lower_bound)
  { for(i=N_BIN_POWERS_OF_TEN-1,
        bitpos = 1U<<(N_BIN_POWERS_OF_TEN-1);
        bitpos; i--, bitpos>>=1)
      { double q;
        q = d0 * bin_exponents_table[i];
        if (q < upper_bound)
          {
            d0 = q;
            exponent -= bitpos;
          }
      }
  }
else
  { for(i=N_BIN_POWERS_OF_TEN-1,
        bitpos = 1U<<(N_BIN_POWERS_OF_TEN-1);
        bitpos; i--, bitpos>>=1)
      { double q;
        q = d0 * bin_fracs_table[i];
        if (q >= lower_bound)
          { d0 = q;
            exponent += bitpos;
          }
      }
  }
exponent += precision;

int imant = (int)d0;

// Decimal point will only move a certain distance:
// outside that range force scientific form.
if (exponent> precision || exponent <0) scientific_form = 1;

int enable_trailzero_supress = 0;
for (int p = precision; p >= 0; p--)
  { int d = 0;
    while (imant>=dec32_table[p])
      { imant -= dec32_table[p];
        d++;
      }
    putchar(d + '0');
    if (enable_trailzero_supress && imant == 0) break;
    if (p == precision + (scientific_form ? 0: -exponent))
      { putchar('.'); // Print decimal point.
        enable_trailzero_supress = 1;
      }
  }
if (scientific_form) // Print the exponent
  { putchar('e');
    bin2ascii(exponent);
  }
}
```

```c
// Scale mantissa to a suitable value for casting to an integer.
int exponent = 0;
unsigned int i,
bitpos = 1U<<(N_BIN_POWERS_OF_TEN-1);
int lower_bound = tens_table[precision];
int upper_bound = tens_table[precision+1];
if (d0 < lower_bound)
  { for(i=N_BIN_POWERS_OF_TEN-1, bitpos = 1U<<(N_BIN_POWERS_OF_TEN-1);  bitpos; i--, bitpos>>=1)
      { double q;
        q = d0 * bin_exponents_table[i];
        if (q < upper_bound)
          {
            d0 = q;
            exponent -= bitpos;
          }
      }
  } else
  { for(i=N_BIN_POWERS_OF_TEN-1, bitpos = 1U<<(N_BIN_POWERS_OF_TEN-1); bitpos; i--, bitpos>>=1)
      { double q;
        q = d0 * bin_fracs_table[i];
        if (q >= lower_bound)
          { d0 = q;
            exponent += bitpos;
          }
      }
  }
exponent += precision;
```

```
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) ];



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. *)
```

```
        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


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")]
```

Floating point representation

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.

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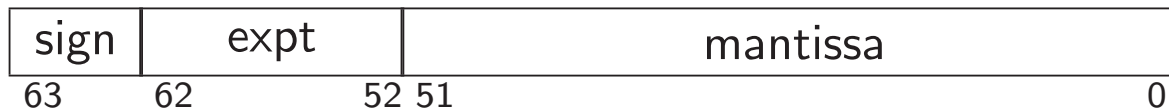
Single precision: 32 bits (1+8+23), $\beta = 2, p = 24$

sign	expt	mantissa

31   30   23 22   0
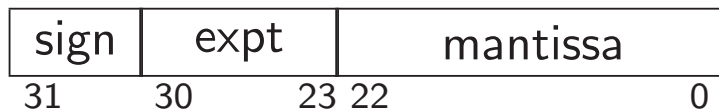
Value represented is *typically*: $(s\ ?\ -1 : 1) * 1.mmmmmm * 2^{eeeee}$.

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

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.

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)

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.

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).

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

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

'*denormalised numbers*' can range down to $2^{-150} \approx 1.401298\text{e-}45$, 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 ...).

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

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

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

So you mean 0.1 is not exactly representable?

Its nearest single precision IEEE number is the *recurring* 'decimal' 0x3dccccd (note round to nearest) i.e. | 0 | 011 1101 1 | 100 1100 1100 1100 1100 1101 |.

Decoded, this is $2^{-4} \times 1.100\,1100 \cdots 1101_2$, or

$$\tfrac{1}{16} \times (2^0 + 2^{-1} + 2^{-4} + 2^{-5} + \cdots + 2^{-23}).$$

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.

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 0.11001100110011
  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`

BTW, So how many times does

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

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

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

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

The examples are on the web site both in C and Java.

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 $10^7 \leq 2^{24} \leq 10^8$.

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

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

Signed zeros 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:

```c
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 ?

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.

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).

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

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

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

IEEE 754 and Intel x86

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

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

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

Floating point operations

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]

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 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.

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

When we do a floating point computation, 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).

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]

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

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

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

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

Errors 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).
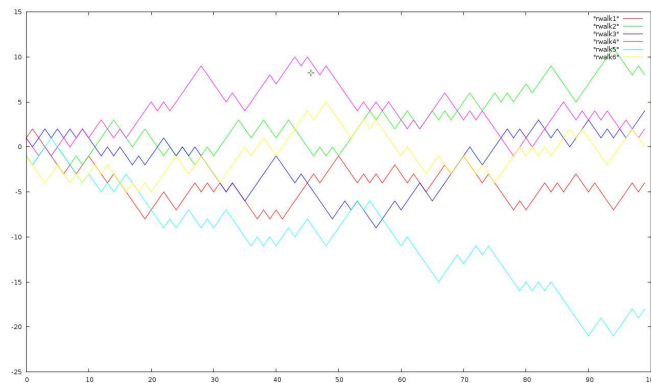
Growth of relative error in p:

```
float p = 1.0, x = x̂ + 0.0;

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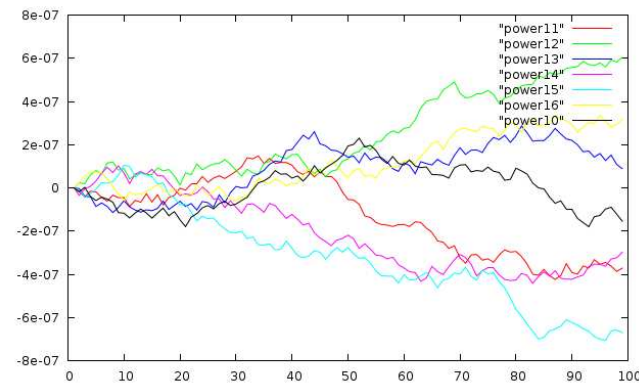

If the steps are uncorrelated: expected sum is zero.

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

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

Summations 100 random values each of ± 1
N=100, a=1. $E[|\sum X|] = a\sqrt{N}.$
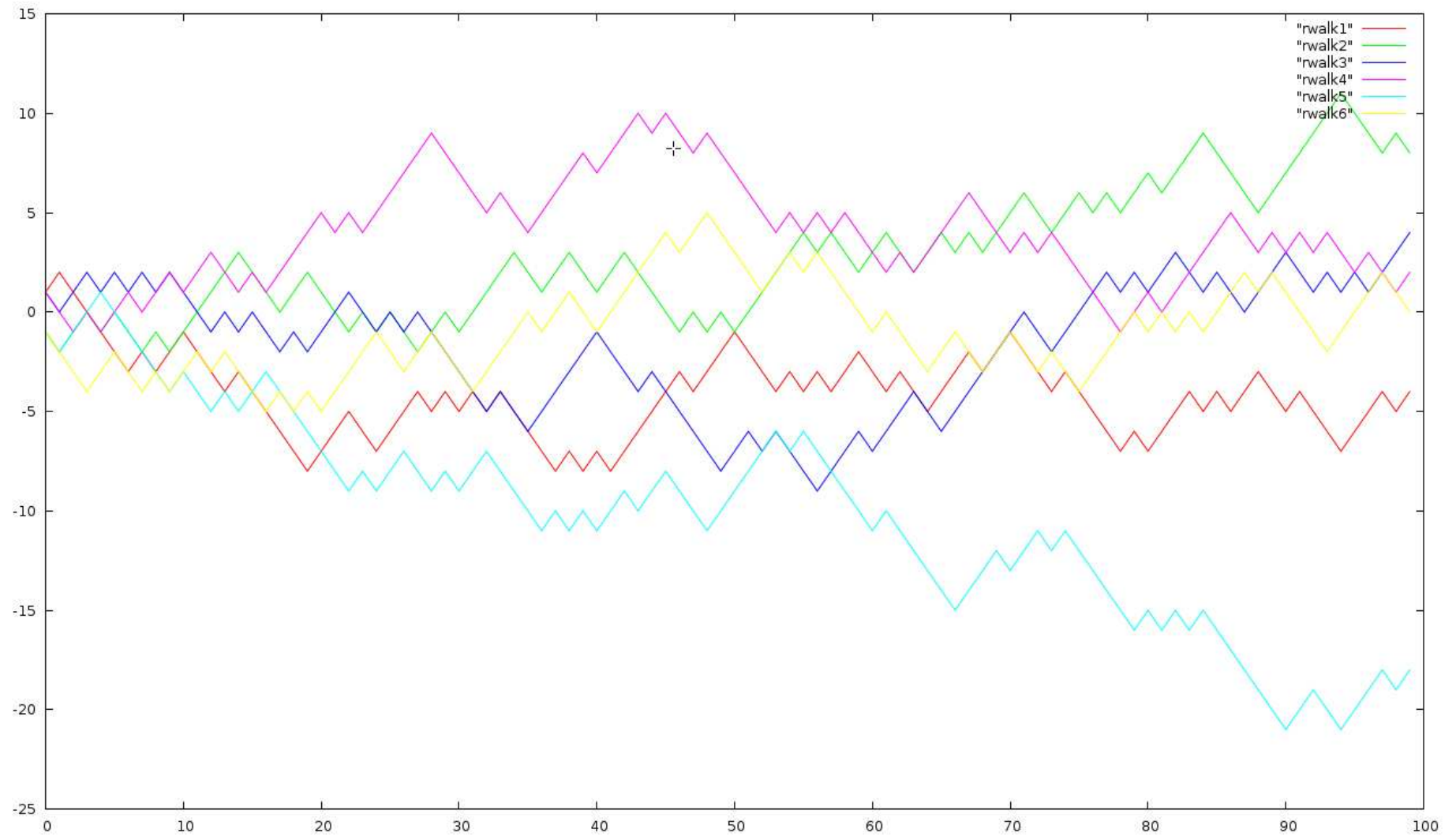
```
float p = 1.0, x = x̂ + 0.0; for (i=0;i<100;i++) { p=p * x; }
```
Growth of relative error in p for various **representable**
values of x:
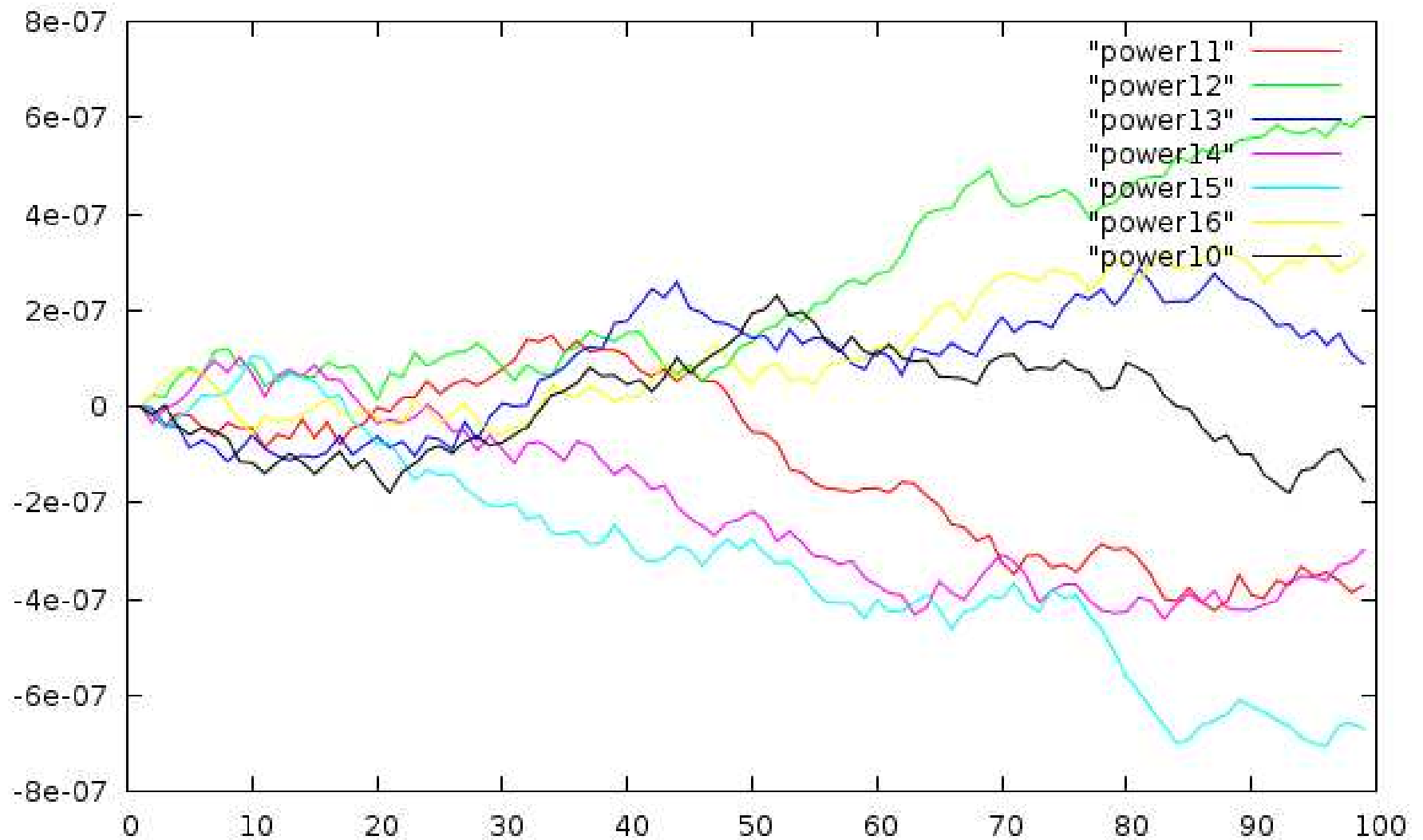
```
float p = 1.0, x = x̂±ε ; for (i=0;i<100;i++) { p=p * x; }
```
Growth of relative error in p for various **unrepresentable** values of x:
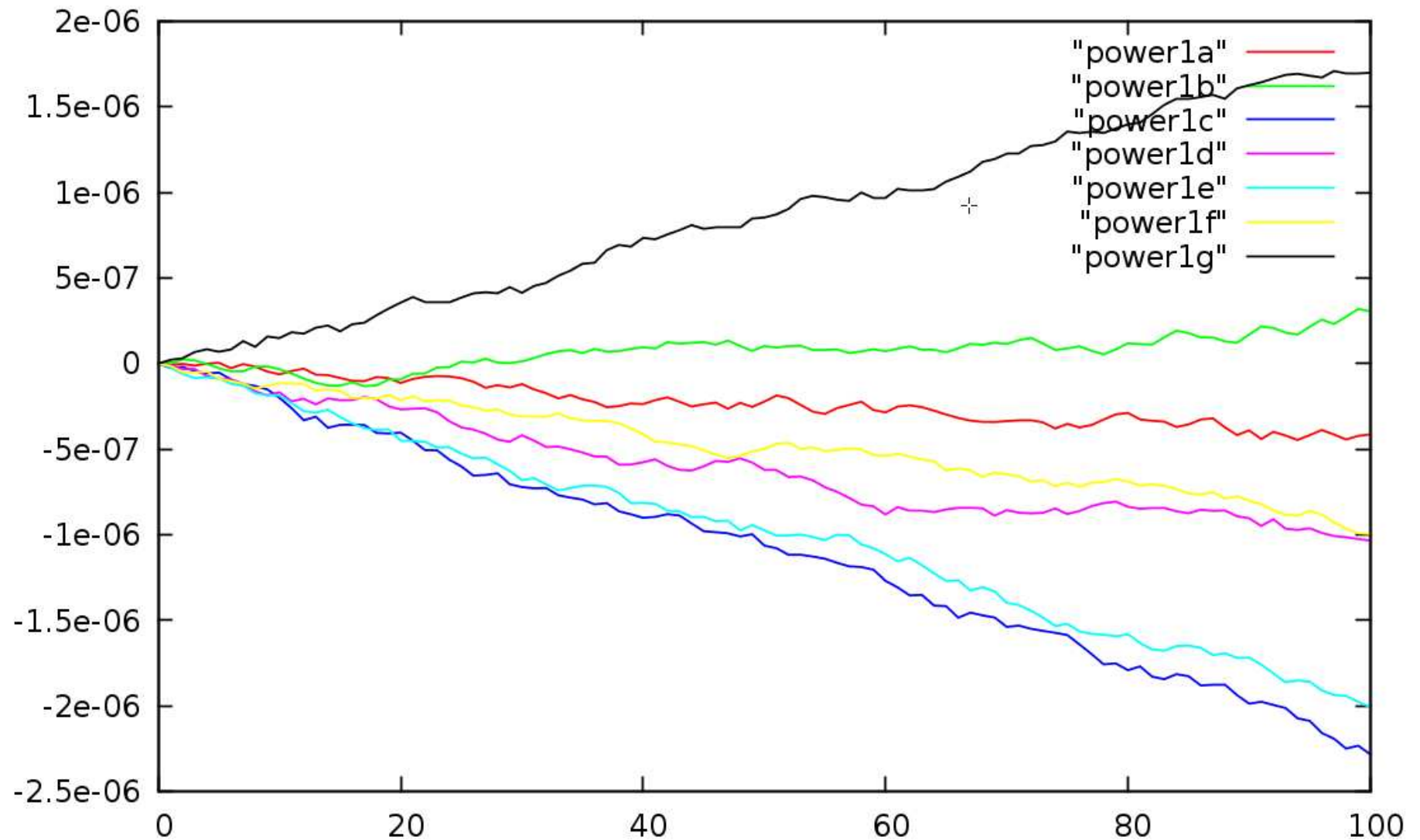
Growth of relative error in p:

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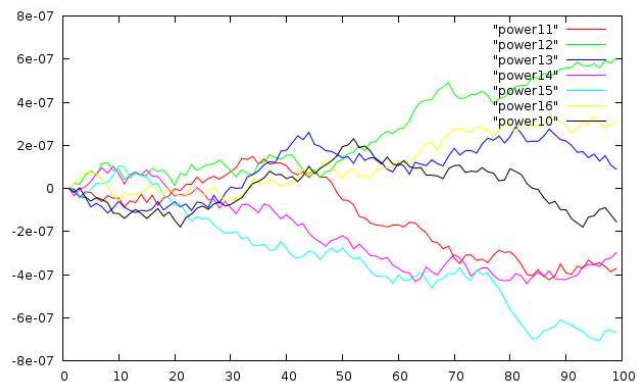
The same computation

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

($\hat{x}$ is not representable).
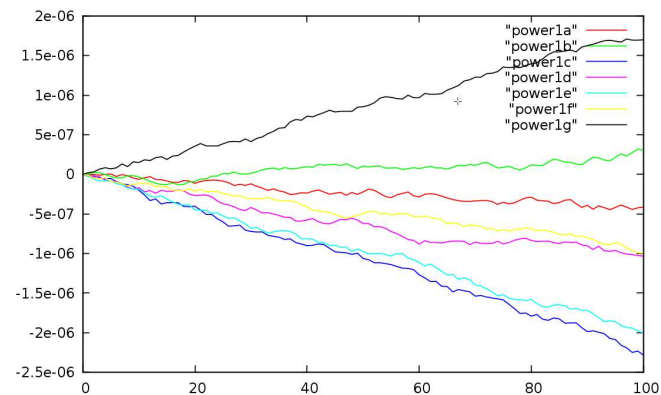
The first plots were for when x_k is exactly representable $(x_k = \hat{x_k})$.

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

Simple worst-case analysis dominates random walk model.

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 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.

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.

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 is useful as it gives an upper bound on the relative error caused by getting a floating point number wrong by $1\,\mathrm{ulp}$, and is therefore useful for expressing errors independent of floating point size.

Floating point ($\beta = 2, p = 3$) numbers: macheps=0.25
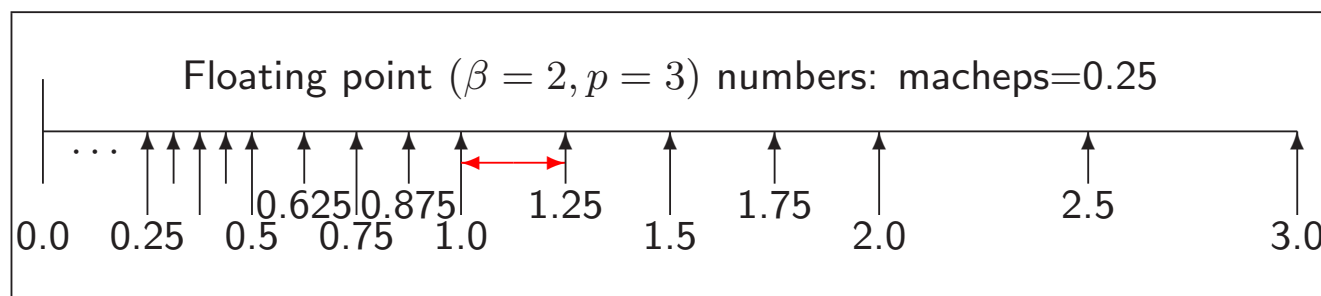
(The relative error caused by being wrong by $1\,\mathrm{ulp}$ *can* be up to 50% smaller than this, consider 1.5 or 1.9999.)

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
```

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.

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 $2^0 \times 1.0001$.

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

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

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

Simple maths, simple programs

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.

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.

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?

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 $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$.

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

$$= \quad \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}}$$

$$= \quad \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$

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

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

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

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


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


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


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


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

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

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

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

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

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

But is this correct? Is it the only answer?

[See SumFwdBack.java on the course website for a program.]

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: 13.57887778897524611

- backward: 13.5788777897519921

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

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.

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`

[See SumFwdBack.java on the course website.]

Infinitary/limiting computations

Rounding versus Truncation Error

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

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

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

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?

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.'

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.

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

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

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

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

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

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

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.

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

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

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!

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 linear convergence.

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:

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

I.E. first-order convergence.

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

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

Nomenclature:

When $\epsilon_{n+1} = k\epsilon_n$ we say that the iteration exhibits "first-order convergence".

For the iteration $x_{n+1} = f(x_n)$ then k is merely $f'(\sigma)$ where σ is the solution of $\sigma = f(\sigma)$ to which iteration is converging.

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

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 Golden.java and GoldenBad.java]

Golden ratio iteration (5)

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

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

i	x	err	err/prev.err
1	1.6180339887498951	2.2204e-16	
2	1.6180339887498958	8.8818e-16	4.0000e+00
3	1.6180339887498978	2.8866e-15	3.2500e+00
4	1.6180339887499045	9.5479e-15	3.3077e+00
5	1.6180339887499260	3.1086e-14	3.2558e+00
6	1.6180339887499957	1.0081e-13	3.2429e+00
7	1.6180339887502213	3.2641e-13	3.2379e+00
	...		
32	3.9828994989829472	2.3649e+00	3.8503e+00
33	14.8634884189986121	1.3245e+01	5.6009e+00
34	219.9232879817058688	2.1831e+02	1.6482e+01

An iteration may enter an infinite loop called a **Limit Cycle**.
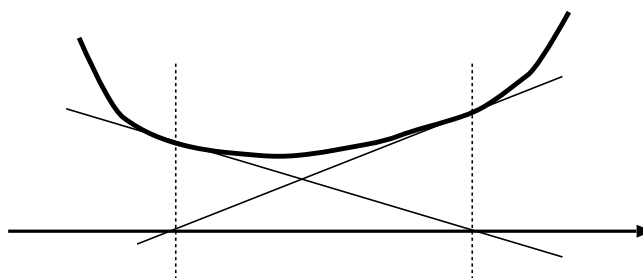
1. This may be mathematically correct (e.g. Newton iteration of a pseudo root where the function just fails to kiss the x-axis).

2. It may arise from discrete (and non-monotone) floating point details.

Where is my beautiful root?
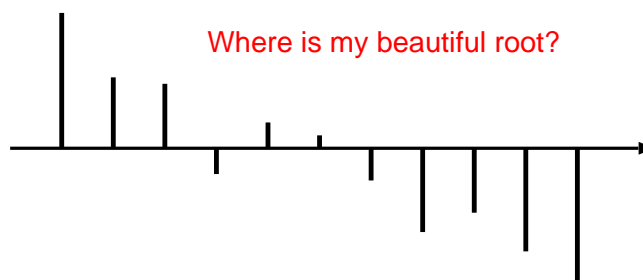
Termination Condition Summary

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 root 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.

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.

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

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

This is *quadratic* convergence.

Pros and Cons:

- Quadratic convergence means that the number of decimal (or binary) digits doubles every iteration

- Problems if we start with $|f'(x_0)|$ being small

- (even possibility of looping)

- Behaves badly near multiple roots.

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.
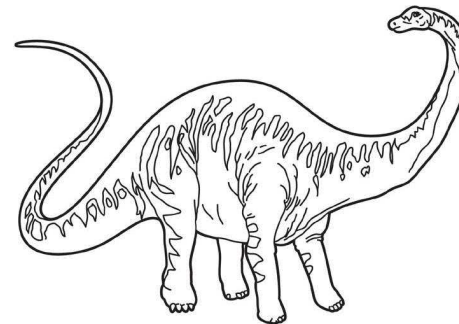
Very big in the middle but ever-so-much smaller at each end!

[See `SinSeriesFloat.java`.]

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.

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

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

then the first missing term will by in x^{n+1} (or x^{n+2} for $\sin(x)$).

This will be the dominant error term (i.e. most of the truncation error), at least for small x.

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

Information transform theory tells us we can fit an nth 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 linear combination

$$L(x) = \sum_{j=0}^{k} y_j \ell_j(x)$$

$$\ell_j(x) = \prod_{0 \leq m \leq k}^{m \neq j} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)},$$

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 Matlab/Octave code `poly.m`.

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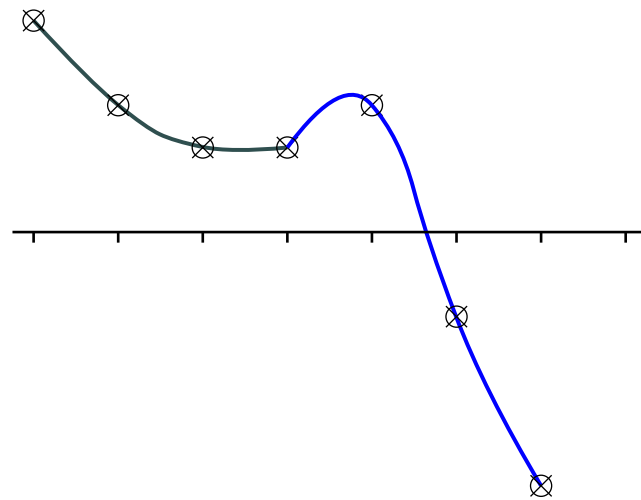
Fitting cubics through four points with one point shared gives a continuous function with a sharp corner!

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.

Quadrature Methods

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 a spline to the data and find the area under the spline.

Commonly used techniques (A-level syllabus?) are :

1. **Mid-point Rule** - puts a horizontal line at the top of each strip,

2. **Trapezium Rule** - uses an appropriate gradiant straight line through the top corners of each strip,

3. **Simpson's Rule** - fits a quadratic to the top of each strip,

4. and clearly higher-order techniques are a logical extension...

Note: These topics may be mentioned elsewhere in these notes and in lectures but this slide was not formally lectured and are not examinable this year.

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

If we fit an 11th-order poly-nominal through the 11 points $\pm 5, \pm 4...0$ we see large errors.

Dashed is the function.

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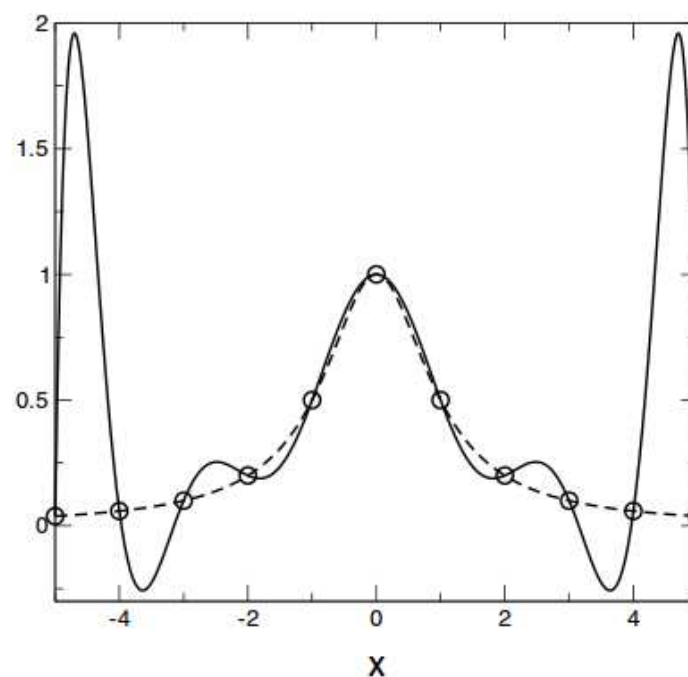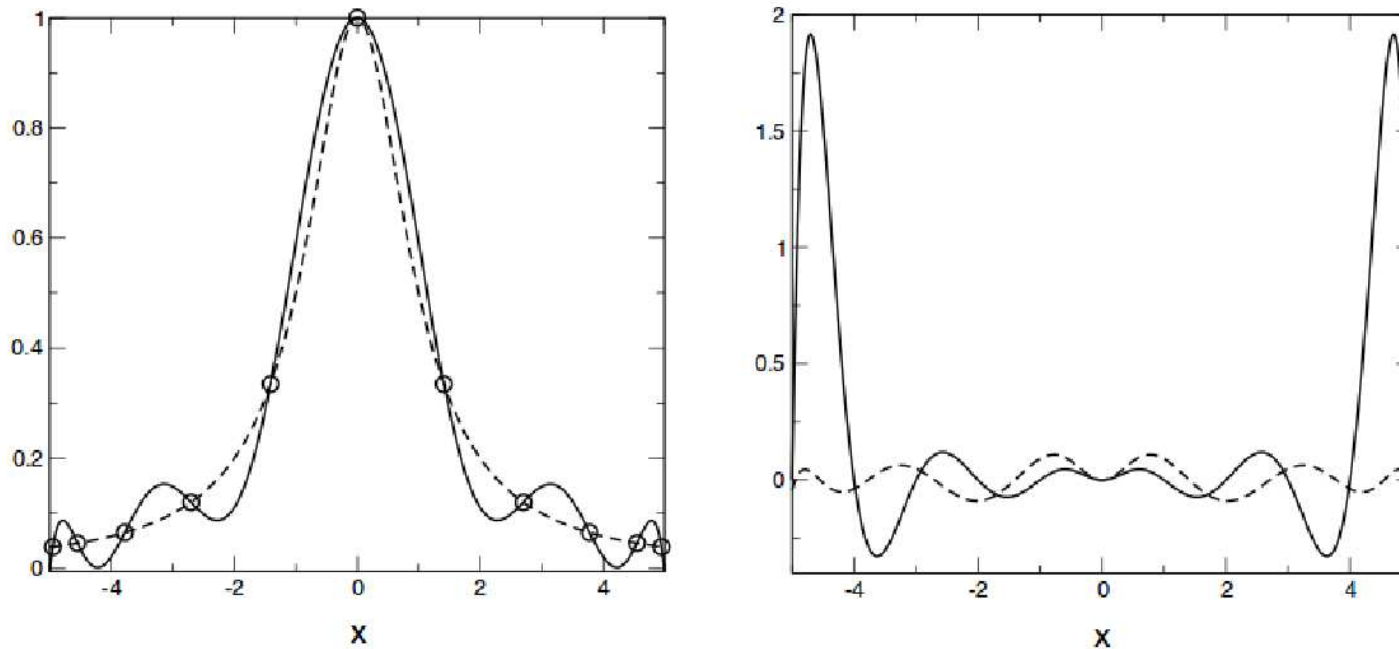
Figure 3.2. *Left: the function* $f(x) = 1/(1 + x^2)$ *is plotted together with the interpolation polynomial for the 11 Chebyshev points* *Right: the interpolation errors for equispaced points and Chebyshev points is shown.*

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.

Pafnuty Lvovich Chebyshev (1821 – 1894).

ВЕЛИКИЙ
РУССКИЙ МАТЕМАТИК

1821   1946

П.Л. ЧЕБЫШЕВ

ПОЧТА 60 СССР
КОП

Hmmm! Someone either lived a long time or else some-one else was not good with figures!.

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:

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

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

ML Coding.

```
(* ML coding of Tchebyshev basis functions *)
fun compute_cheby n x =
    let fun tch 0 = (1.0, 123456.0) (*snd arbitrary*)
        |    tch 1 = (x, 1.0)
        |    tch n =
                let val (p, pp) = tch (n-1)
                in (2.0 * x * p - pp, p) end
    val (ans, _) = tch n
    in ans end
;


(* crude mid-point rule numeric integration to get nth coef for expansion of f. *)
fun cheby_coef n f =
    let val h = 0.00001
        fun range2 x ev (delta:real) =
                if x>ev then [] else x :: (range2 (x+delta) ev delta)
        val points = (range2 (~1.0+h) (1.0-h) h)
        fun compute (x,c) => f x  * compute_cheby n x / Math.sqrt(1.0 -x*x) + c)
        val sum = foldl compute 0.0 points
        val ans = sum * h  * 2.0 / Math.pi
    in if n=0 then ans/2.0 else ans
    end
```

Some plots: Two of the basis functions:
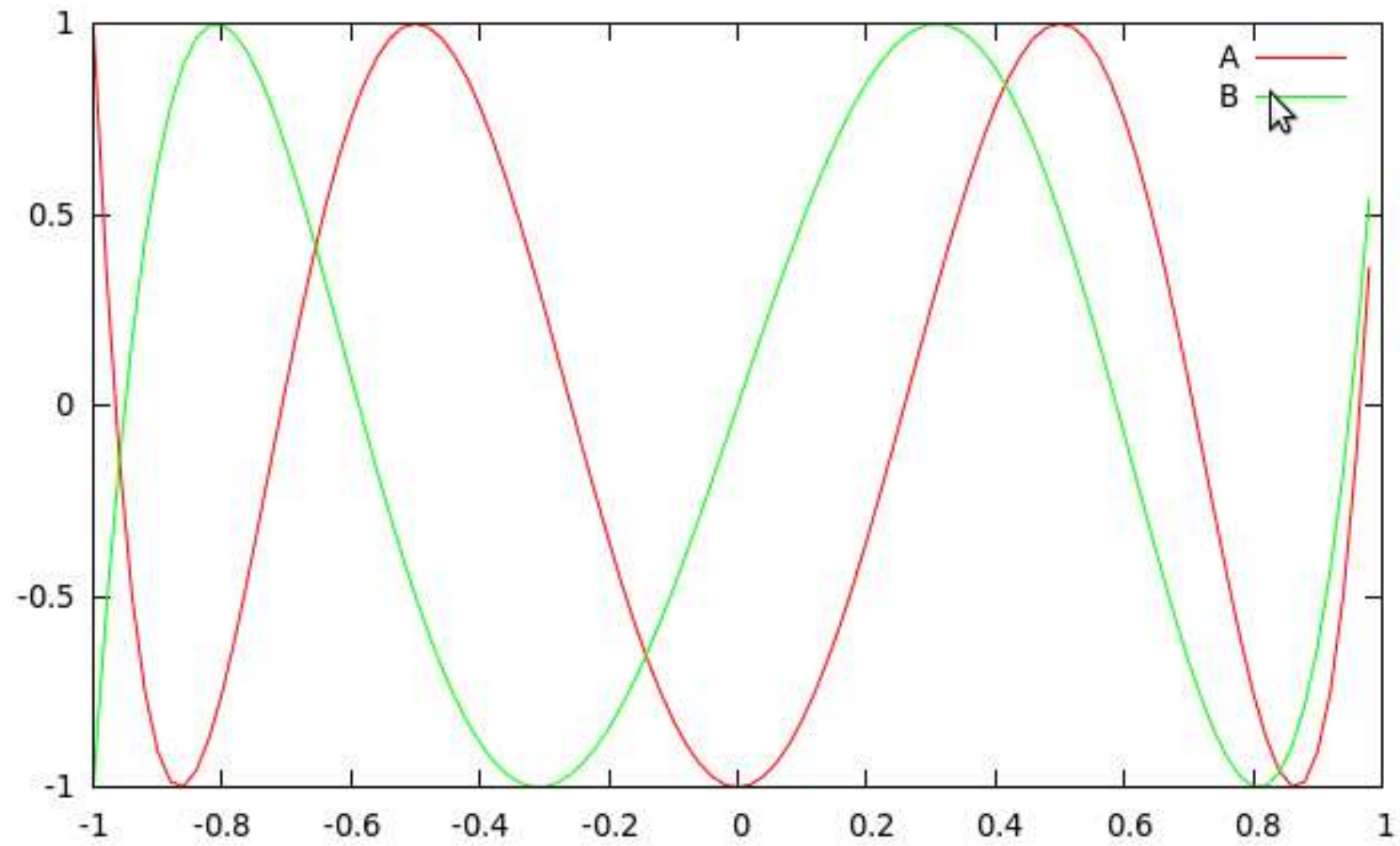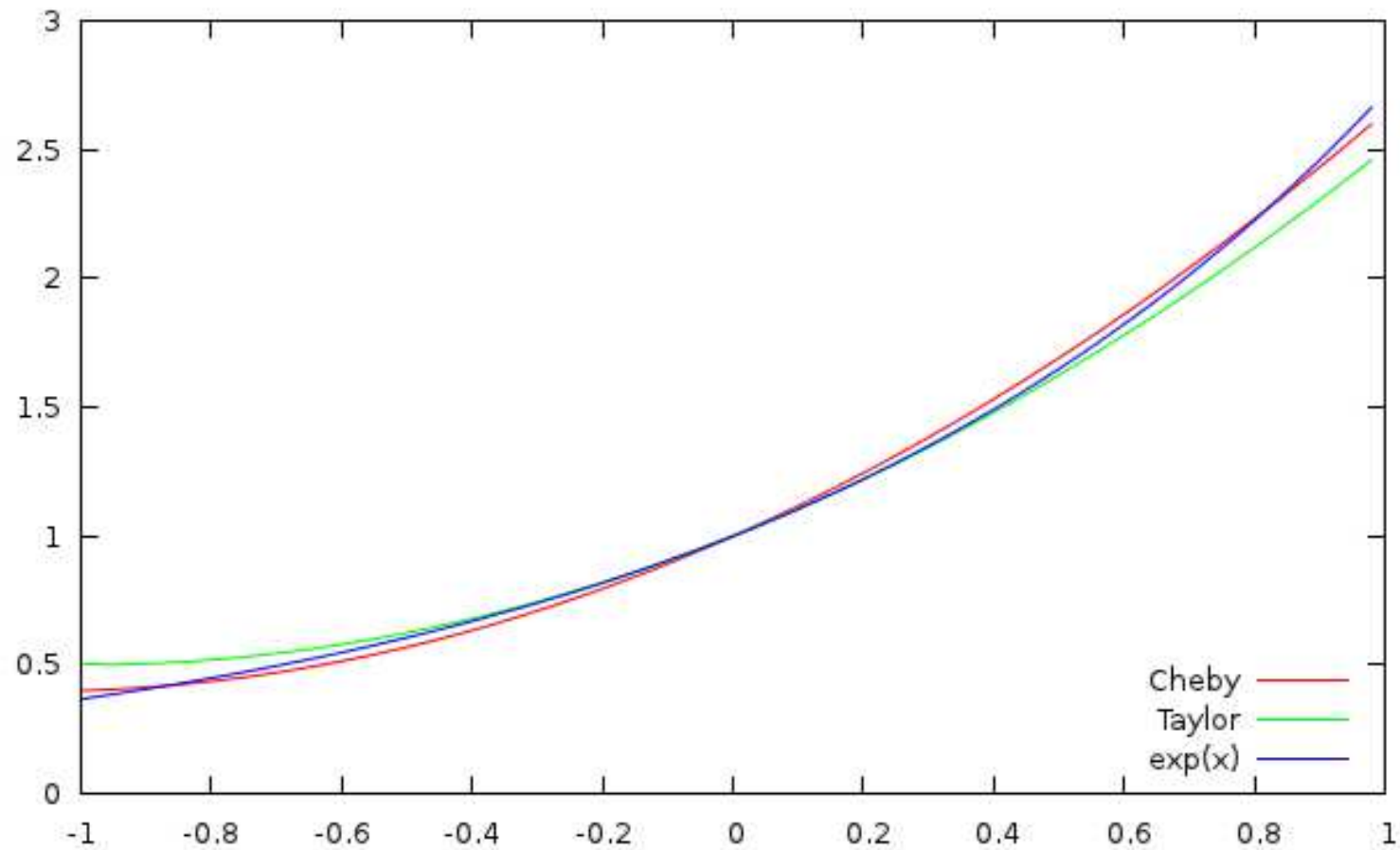
Both Taylor and Cheby deploy a quadratic but Chebby fits much better in this interval.

Interpolating log(1+x) using quartics.
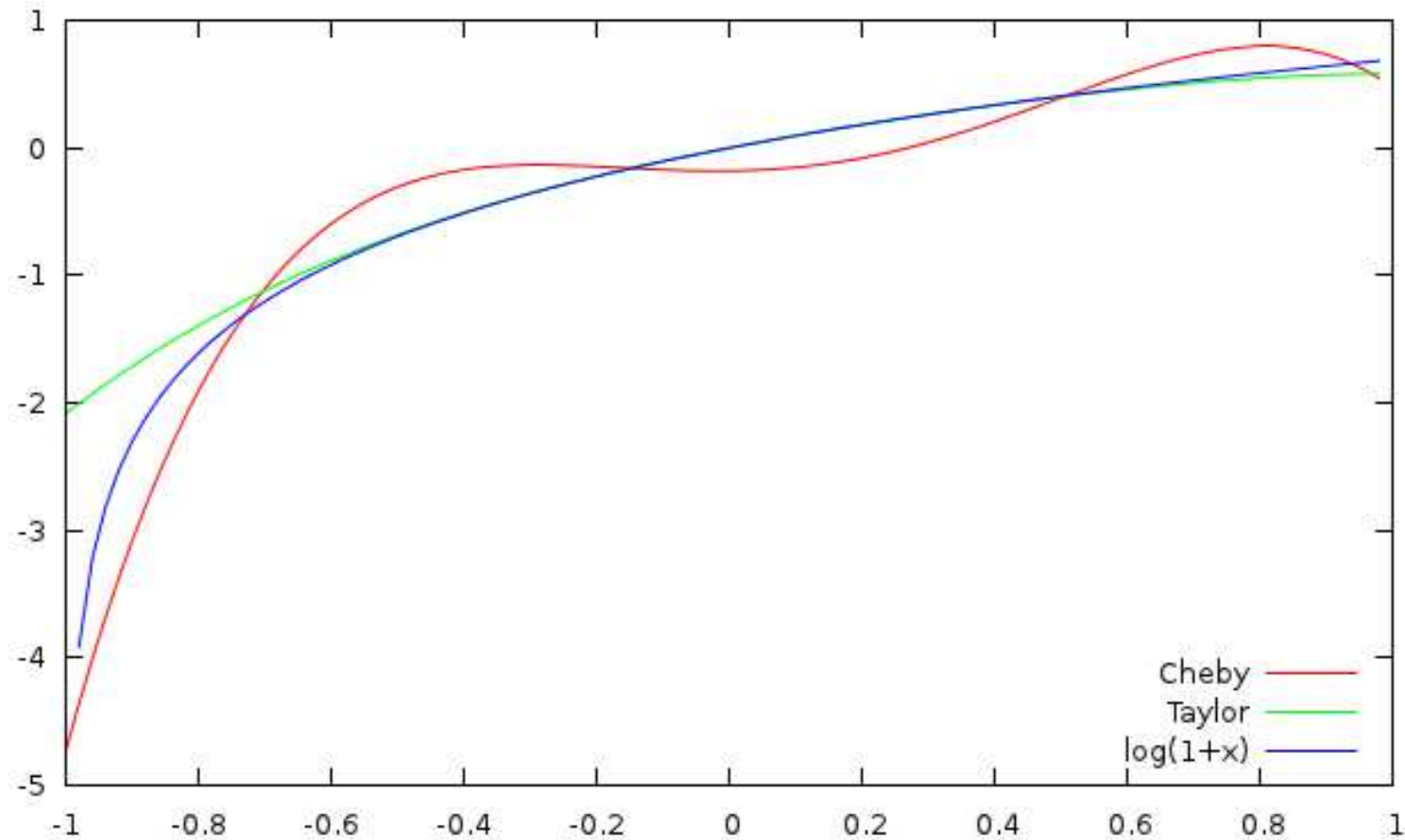
In this plot we used quartics.

Taylor has all its error at the left end while Cheby has
uniform error.

The formula for the Chebyshev coefficient, as in the ml program online, is

$$a_n = \frac{2}{\pi} \int_{-1}^{+1} \frac{f(x)T_n(x)}{\sqrt{1 - x^2}} \, \mathrm{d}x$$

Actually you need half this expression for a_0.

This formula will be provided if used in an examination or exercise.

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.

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

$$
\begin{aligned}
y_0(x) &= (x-1)^7 \\
y_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1
\end{aligned}
$$
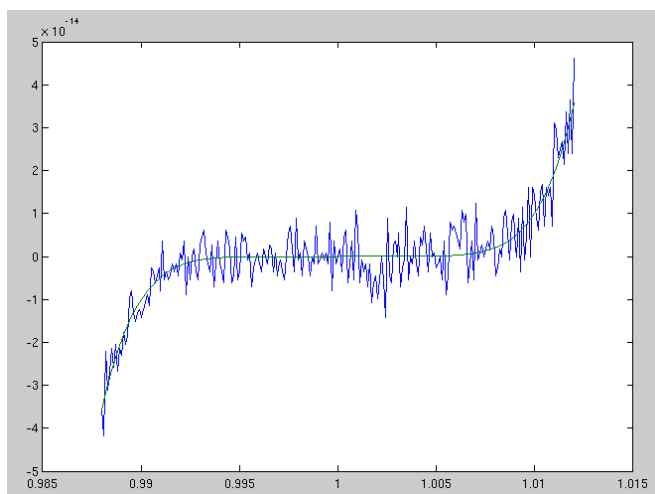
Exercise: compare the roughness in the plot to your own estimate of the absolute error.

```
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.

http://www.hackersdelight.org/

Total ROM used: a mere 320 words.

http://righto.com/sinclair

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.'*

One of Sir Clive Sinclair's Early Products

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 $(1, 0)$.

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

$$\Theta = \sum_i \alpha_i$$

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

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

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.

$$
\begin{pmatrix} 1 & -0.5 \\ 0.5 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix}
$$
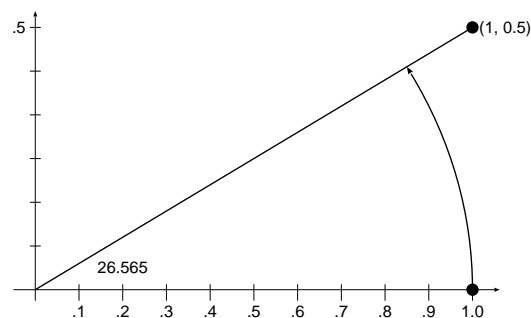
The subdivisions are: $\tan 26.565° \approx 0.5$, $\tan 14.036° \approx 0.25$, ...

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

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 $0.5 > 22.5$? How would you use CORDIC for angles greater than 45 degrees?

CORDIC ML implementation:

```
fun cordic loops arg =
  let fun iterate n p arg v =
      if n=loops then v
      else
      let
          val (d, sigma) = if arg > 0.0 then (1.0, p) else (~1.0, ~p)
          val rotor = { a= 1.0, b= ~sigma, c= sigma, d= 1.0 }
      in iterate (n+1) (p/2.0) (arg - d * Vector.sub(Angles, n)) (mpx4b2 rotor v)
      end

  val rotated = iterate 0 1.0 arg { re=1.0, im=0.0 }
  in #re rotated * Vector.sub(Kvalues, loops-1) (*Take real part for cosine*)
  end


map (fn x=>(cordic 6 x, Math.cos x)) [ 0.0, Math.pi/4.0, 0.1,  ~ Math.pi/4.0, 1.0 ]
val it =
    [(0.9997434238, 1.0), (0.7074836445, 0.7071067812),
     (0.9941095003, 0.9950041653), (0.7074836445, 0.7071067812),
     (0.5624471808, 0.5403023059)] : (real * Real.Math.real) list
```

CORDIC C - large font - emphasising no multiplications
needed:

```c
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);    // Divide by 2^k using a right shift.
    int ty = y + (((x>>k) ^ d) - d);
    theta = theta - ((cordic_ctab[k] ^ d) - d);
    x = tx; y = ty;
  }
  printf("Ans=(%i,%i)/10^9", x, y);
}
```

// Two's complement negation : flip all bits (by xor with
-1) and add 1 (by subtracting -1). Avoids conditional branch.

CORDIC Java - large font - emphasising no multiplica-
tions needed:

```java
public static double cordic(double theta)
{
  int x=607252935 /*prescaled constant for iterations=32*/, y=0;
  for (int k=0; k<iterations; ++k)
  {
    int tx, ty;
    if (theta >= 0) // All nice simple integer arithmetic
      { tx = x - (y >> k);
        ty = y + (x >> k);
        theta -= ctab[k];
      }
    else
      { tx = x + (y >> k);
        ty = y - (x >> k);
        theta += ctab[k];
      }
    x = tx; y = ty;
  }
  return (double)x; // Returns cosine * 10^9.
}
```

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.

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.

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:

```
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.

. . . 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.

- `float` is very much a second class type like `char` and `short`.

- Constants `1.1` are type `double` unless you ask `1.1f`.

- `float`s are implicitly converted to `double`s at various points (e.g. for 'vararg' functions like `printf`).

- The ISO/ANSI C standard says that a computation involving only `float`s may be done at type `double`, so f and g in

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

  may give different results.

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

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.

Some nastier issues

Consider solving

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

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.

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*.

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.

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.

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

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

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

E.g.

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

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

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

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} !!

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.

Multiplication by a precisely-represented, large constant gives a large first derivative but the relative error is unchanged. Hence it is the change in relative error that is important, not in magnitude.

When there are multiple inputs and outputs we normally quote the worst condition number from any input to any output.

A problem with a low condition number is said to be well-conditioned.

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

$$f_{(\text{with condition number } 5)}(1.23456789) = n.pqr$$

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

Condition Number: Simple Example(s)

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...

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.

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 for 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 few percent. Solve all these.

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

Adaptive Methods

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".

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

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 later.]

Mandelbrot set
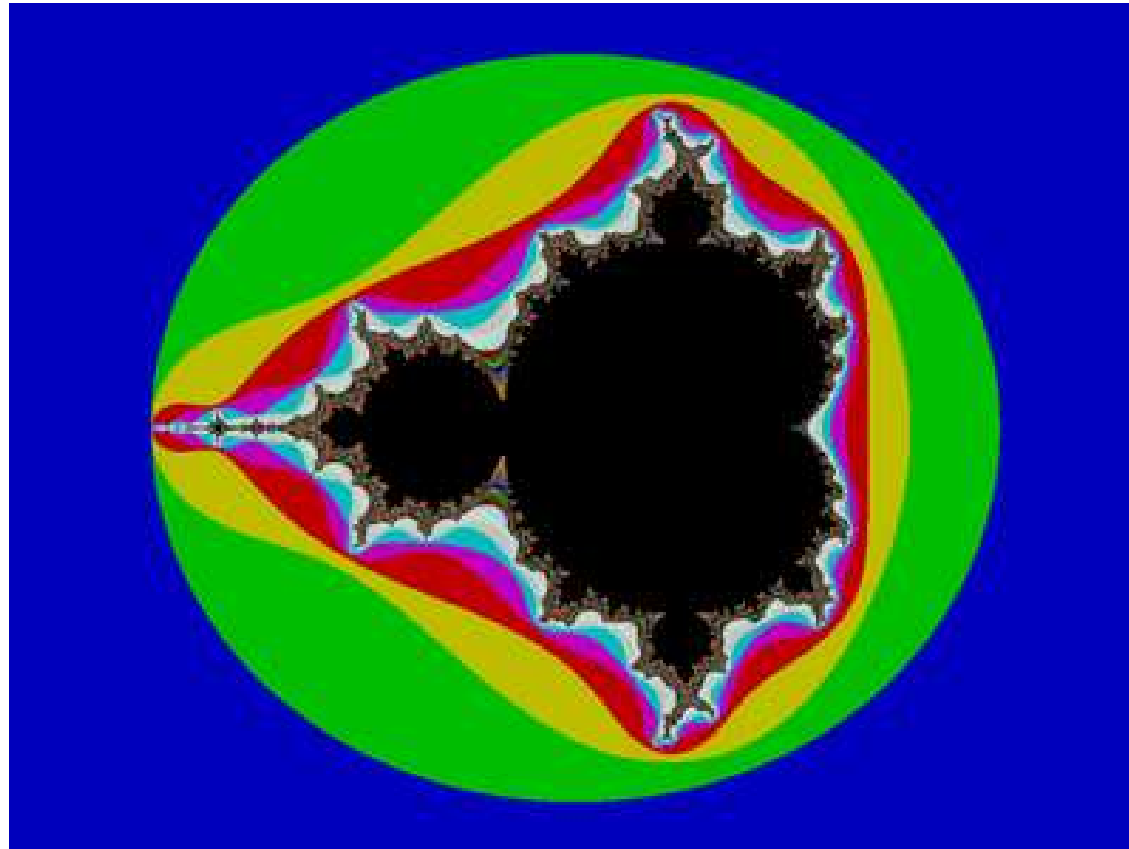
Solving Systems of Simultaneous Equations

Solving Linear Simultaneous Equations

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

<div align="center">We wish to find $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$.</div>

This can be found directly by finding the inverse

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

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 $-3X_1 - 6 = -3 \rightsquigarrow X_1 = -1$.

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

Complexity is: $O(n^3)$

Gaussian Elimination Minor Problems

The very first step is to multiply the second 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 pivoting**.

Unless all potential pivots are small we will not commonly use **full pivoting**. This is where, we also permute columns. Column permutation requires we keep track of the new X_i variable ordering.

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 it better

to 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.

```java
public static void MakeUpperTri(double [][] Adata, boolean pivot_enable)
// Note, this is not full L/U decomposition since we throw away the info to make L.
{ for (int k=0; k<Adata.length; k++)
  {
    if (pivot_enable)
     { double p = 0.0;
       int k1 = 0;
       for(int i=k; i<Adata.length; i++)
         if (Math.abs(Adata[i][k]) > p) // Pivoting : find largest
          { p = Math.abs(Adata[i][k]);
            k1 = k;
          }
       double [] t = Adata[k]; Adata[k] = Adata[k1]; Adata[k1] = t;
     }
    for (int i=k+1; i<Adata.length; i++)
      {
        assert(Adata[k][k] != 0.0); // Singular matrix!
        double mu = Adata[i][k]/Adata[k][k];
        Adata[i][k] = 0.0;
        for (int j=k+1; j<Adata[i].length; j++)
            Adata[i][j] = Adata[i][j] - mu *Adata[k][j];
      }
  }
}
```

```
// Forwards substitution to solve Ly = b.

public static double [] FwdsSubst(double [][] LL, double [] b)
{   double [] y = new double[LL.length];
    y[0] = b[0];
    for (int i=1; i < LL.length; i++)
      {   double sum = 0.0;
          for (int j=0; j < i-1; j++) sum += LL[i][j] *  y[j];
          y[i] = b[i] - sum;
      }
   return y;
}


// Back substitution to solve Ux = y.

public static double [] BackSubst(double [][] UU, double [] y)
{ double [] x = new double[UU.length];
  for (int i=UU.length-1; i >= 0; i--)
    { double sum = 0.0;
      for (int j=i+1; j < UU.length; j++) sum += UU[i][j] * x[j];
       x[i] = (y[i] - sum)/UU[i][i];
    }
   return x;
}
```

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 your steps 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:
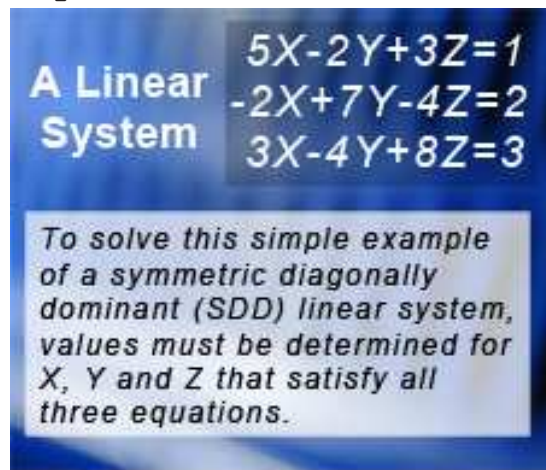
Then $L = \prod_i^{1..N} G_i$
One can form L in place starting from I.

For $1 \leq i, r, c \leq N$

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

Complexity: TBA

A Linear System

$5X-2Y+3Z=1$
$-2X+7Y-4Z=2$
$3X-4Y+8Z=3$

To solve this simple example of a symmetric diagonally dominant (SDD) linear system, values must be determined for X, Y and Z that satisfy all three equations.

The solution is
$X=1/17, Y=12/17, Z=12/17$

Symmetric diagonally dominant: SDD: the magnitude of the diagonal entry is $>= \sum$ magnitudes of all the other row entries.

"2010: Miller, Coutis & Peng of Carnegie Mellon University devised an innovative and elegantly concise algorithm that can efficiently solve systems of linear equations that are critical to such important computer applications as image processing, logistics and scheduling problems, and recommendation systems.

Recommendation systems, such as the one used by Netflix to suggest movies to customers, use SDD systems to compare the preferences of an individual to those of millions of other customers."

Non-examinable

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:

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}$$

$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{j,k} \right), \text{ for } i > j.$$

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...

Java: Cholesky Transform for Symmetric +ve-definite Matrices

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}$$

$$L_{i,j} = \frac{1}{L_{j,j}}\left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k}\right), \text{ for one side only.}$$

```java
public static double[][] Cholesky(double[][] Adata)
{ int m = Adata.length;
  double[][] R = new double[m][m]; //Default 0 for unassigned upper triangle
  for(int i = 0; i<m ;i++)
    for(int j = 0; j < i+1; j++)    //Fill in diagonal and below only.
      { double sum = 0.0;
        for(int k = 0; k < j; k++) sum += R[i][k] * R[j][k];
        R[i][j] = (i == j) ? Math.sqrt(Adata[j][j] - sum):
                             (Adata[i][j] - sum)/R[j][j];
      }
  return R;
}
```

We can apply this technique even for non-symmetric matrices.

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

Then we correct the result using this identity:

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

Exercise: What happens if Gaussian Elimination is attempted for non-square matrices?

What is a Hermitian Matrix ?

This slides for this section discuss square symmetric matrices but they also hold for complex Hermitian matrices.

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

Hermitians are non-examinable.

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

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

This can be a neat solution. Upsides:

- naturally copes with uncertainty in input values

- IEEE arithmetic rounding modes (to +ve/-ve infinity) do much of the work.

This can be a neat solution to some problems. 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.

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

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

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

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

?

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

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.

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

Exact Real Arithmetic (2)

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

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

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

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.

FDTD and Monte Carlo Simulation

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-cell differences are exchanged with neighbours at each step.

Monte Carlo: Either of the above simulators is run with many different random seeds 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 transients).

```
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 $\mathrm{darts} \rightsquigarrow \inf$

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

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

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

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

```
public static int Main()
{
    DartThrowing sim = new DartThrowing();
    sim.OneRun(0, false);
    sim.OneRun(1, false);
    sim.OneRun(3, false);
    return 0;
}
```

Ratio is 785259/1000000 = 3.141036
Ratio is 785669/1000000 = 3.142676
Ratio is 786428/1000000 = 3.145712

Process make finished

Abscissa is number of darts thrown in a batch.
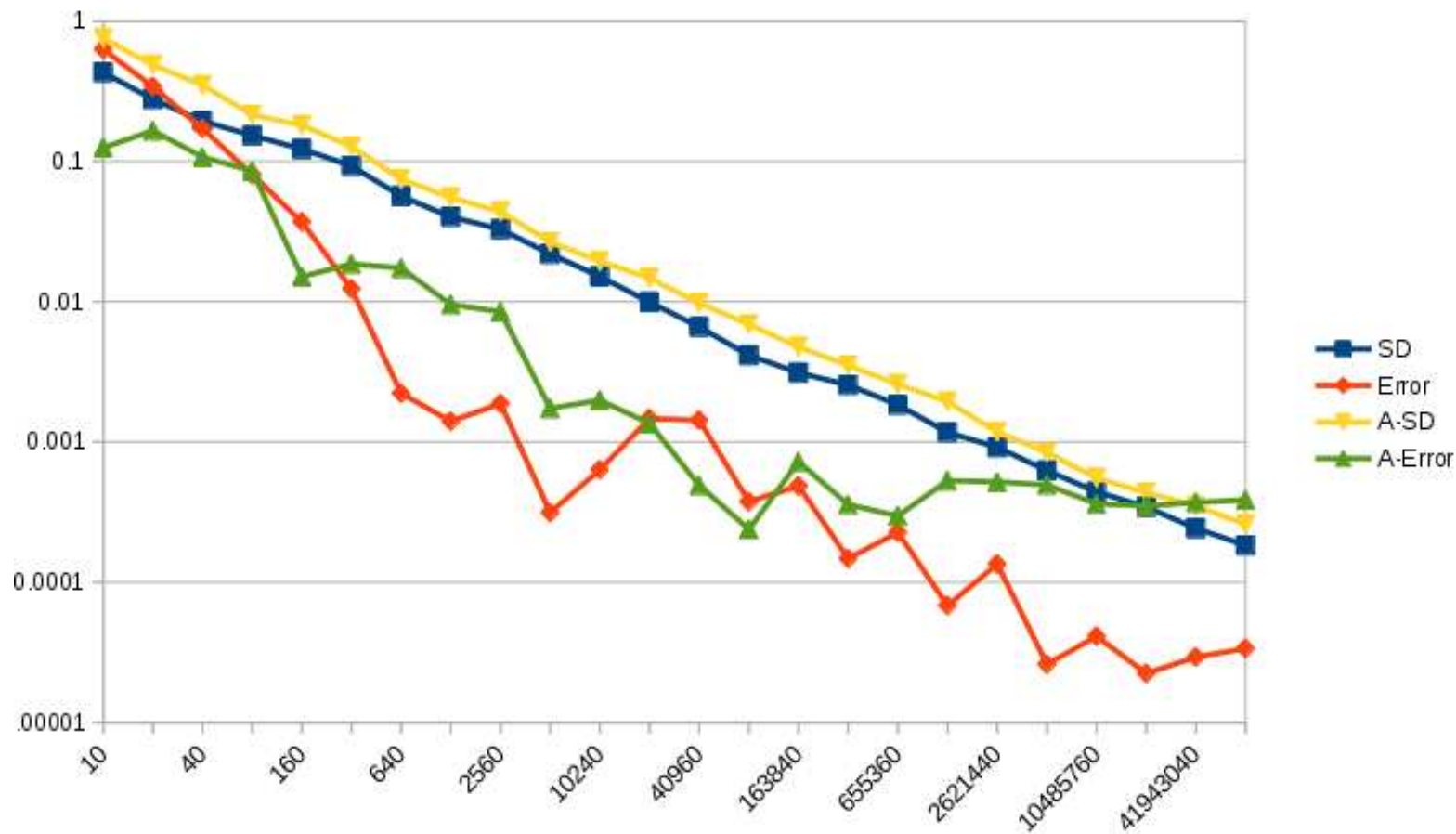
Slow convergence (reciprocal). Adding an **antithetic** dart (non-examinable) (A plots) for each one thrown did not help!

SHM: quadrature oscillator: generates sine and cosine waveforms.

Controlling equations:
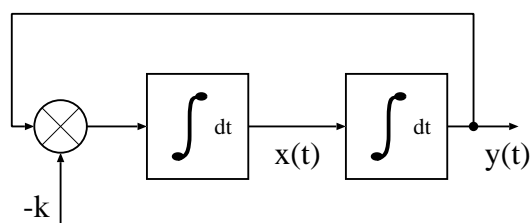
For each time step:

$$x = \int -ky \, \mathrm{d}t$$

$$y = \int x \, \mathrm{d}t$$

$$x := x - ky\Delta t$$

$$y := y + x\Delta t$$

(Example TwoPoleOscillator.cs)

Alternative implementation, using differentiators: is it numerically stable ?

Controlling equations: For each time step:
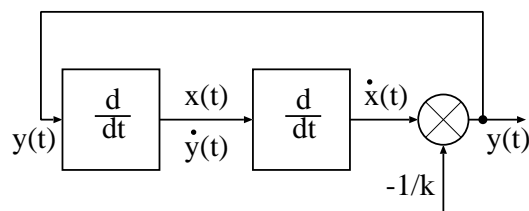
$$\dot{x} = -ky \qquad \dot{x} := (x - x_{\mathrm{last}})/\Delta t$$

$$\dot{y} = x \qquad \dot{y} := (y - y_{\mathrm{last}})/\Delta t$$

$$x := \dot{y} \quad y := -\dot{x}/k$$

(Example shm-integral) Missing! Cannot be made to work?

```csharp
// Initial conditions    (C# code)
x_state = 1.0; y_state = 0.1;
for (int it=0; it<sim_steps; it++)
{
  x_state = x_state + tau * y_state;
  y_state = y_state - tau * x_state;

  x_state = x_state * 0.9999; // Add in some resitive loss

  wf.write_val((int)(x_state * 1000)); // Write to waveform file.

  if (plotting)
  { int x = it / 6, gain = size/10;
    plotter.setget_pixel((uint)x, (uint)(x_state*gain + 2*size/5), false, 0);
    plotter.setget_pixel((uint)x, (uint)(y_state*gain + 3*size/5), false, 128);
  }
  Console.WriteLine(" {0} {1}", x_state, y_state);
}
```

Two physical manifestations of the the same 1-D example:
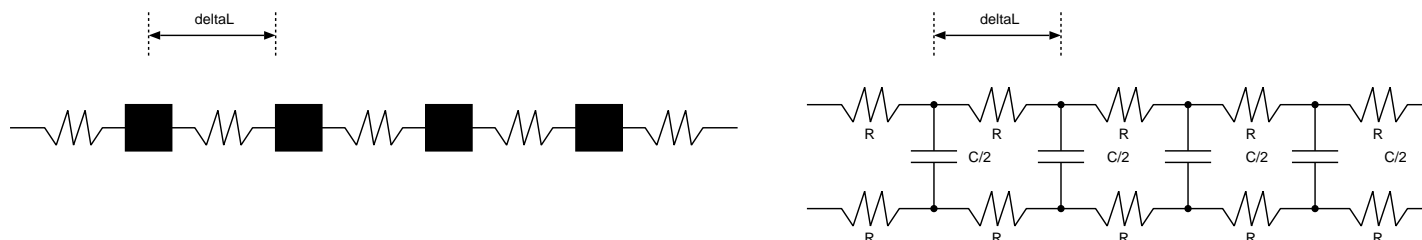
Per element	Heat Conduction in a Rod	R/C Delay Line
$\rho A = dR/dx$	thermal resistance (J/s/$^\circ$C/m)	electrical resistance (Ω/m)
$\epsilon D = dC/dx$	thermal capacity ($^\circ$C/J/m)	electrical capacity (F/m)
Element state	Temperature ($^\circ$C)	Voltage (V)
Coupling	Heat flow (J/s)	Charge flow (Q/s=I)

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

Exercise: Relate these two systems to a third: 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.

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

```
// State Variables
int [] pos = new int [size];
int [] vel = new int [size];

public int RunFDTD(uint sim_steps)
{ if (true)      // Add an initial impulse
  { int o = size/2, ss = size/3;
    for (int x=o-ss; x<o+ss; x++)     pos[x] += 1 * (ss-Math.Abs(x-o));
  }

  for (int it=0; it<sim_steps; it++) // FDTD Simulation loop
  { for (int x=1; x<size-1; x++) pos[x] += vel[x];
    for (int x=1; x<size-1; x++) vel[x] +=  (pos[x-1] + pos [x+1] - 2 * pos[x]);
    int op = pos[64];   // Sample somewhere on the string.
    pos [4] = pos[4] * 96 / 100; // Add a little resistive loss.
    for (int x=0; x<size; x++)
     { const int scale = 4;
       int y = pos[x]/scale;
       plotter.setget_pixel((uint)(x), (uint)(oldy+256), false, 255);
     }
     plotter.setget_pixel((uint)(it/4), (uint)(op/4+100), false, 90);
  }
```

It is handy to have a reference potential.

For the thermal system we might use either 0°C or 0°K.
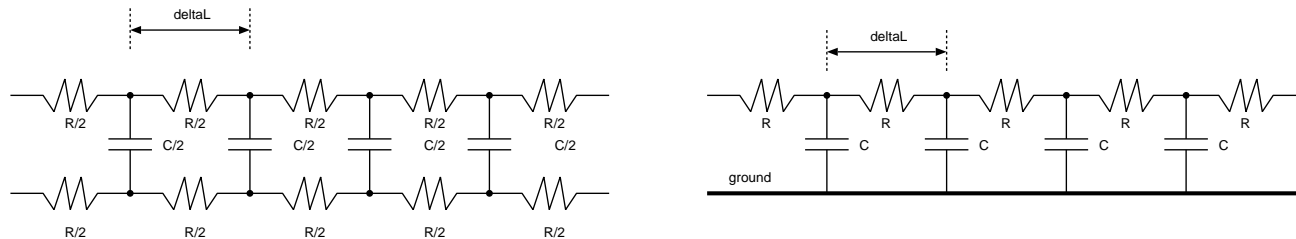
For the electrical system we can put a 'ground plane' down the middle of capacitors. They will now have twice the capacitance and the voltage across them will be half as much.

This slide non-examinable.

Iterating the Finite Differences

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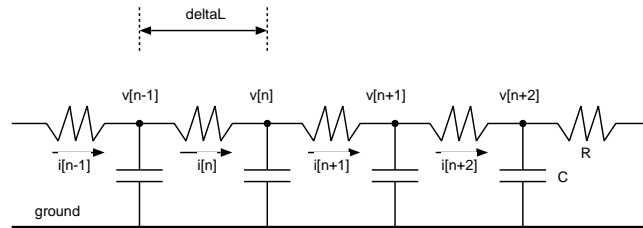
For each time step:

$$i[x] \quad := \quad (v[x+1] - v[x])/R$$

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

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

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

Error problem: 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?

The forward difference method uses the values at the end of the timestep as though constant in the timestep.

End values can (sometimes) be found from simultaneous equations.
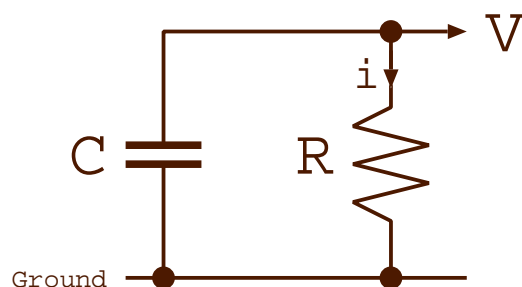
We'll use a very simple example with one state variable: a capacitor discharging or heat dissipation from a blob.

Dynamic equations:

$$dV/dt \;=\; -i/C$$
$$i \;=\; V/R$$

Dyns rewritten:

$$dV/dt \;=\; -\alpha V$$
$$\alpha \;=\; 1/CR$$

Forward iteration:

$$V(n+1) \;:=\; V(n) - \tau V(n)$$
$$\tau \;=\; \Delta T/CR$$

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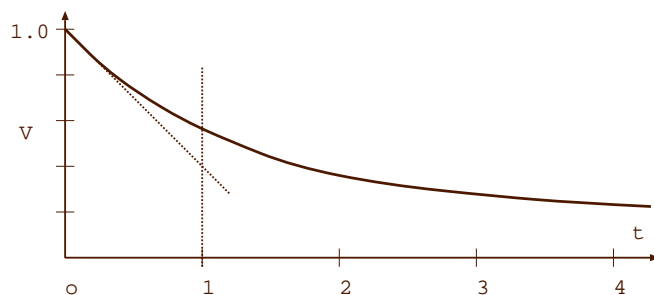
True result given by

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

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

Using starting loss rate throughout timestep leads to a **systematic** overcooling error of 50 ppm.

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

Stability of Interacting Elements.

The forward difference method uses the values at the end of the timestep as though constant in the timestep.

FDTD errors will tend to cancel out if either:

1. they alternate in polarity, or

2. they are part of a negative-feedback loop.

Most systems that are simulated with FDTD have interactions between element pairs in both directions that are part of a stabilising, negative-feedback arrangement. Take an exponential discharge: 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, cancelling the problem to a large extent.

An analysis of this effect is non-examinable this year.

NB: This slide was added after lectures ended to clarify issues arising in the examples sheet.

We can potentially use the ending loss rate in whole time step:
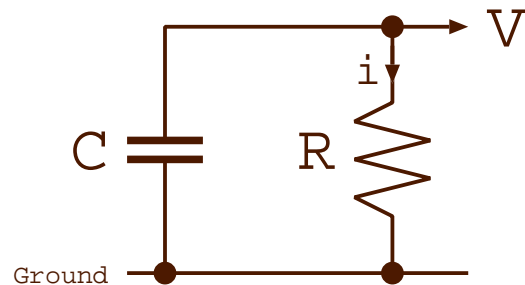
The expressions for the ending values are found solving simultaneous equations. Here we have only one var so it is easy!

Reverse iteration:

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

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**.

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.

These time-domain patterns are known as **stencils**.

Non-examinable. Briefly lectured only.

Techniques with Constant Relative Error
The Further Importance of Logarithms

Weber Fechner Law

"The sensation is proportional to the logarithm of the stimulus."

Ernst Heinrich Weber (1795 — 1878) experimented on his servants and found the **just-noticeable difference between two stimuli is proportional to the magnitude of the stimuli**.

This is the basis for using logarithmic units in everyday measure: e.g. Camera F-stops and measuring sound in decibels.

Used for everyday multiplication and division from 1600 to 1975.
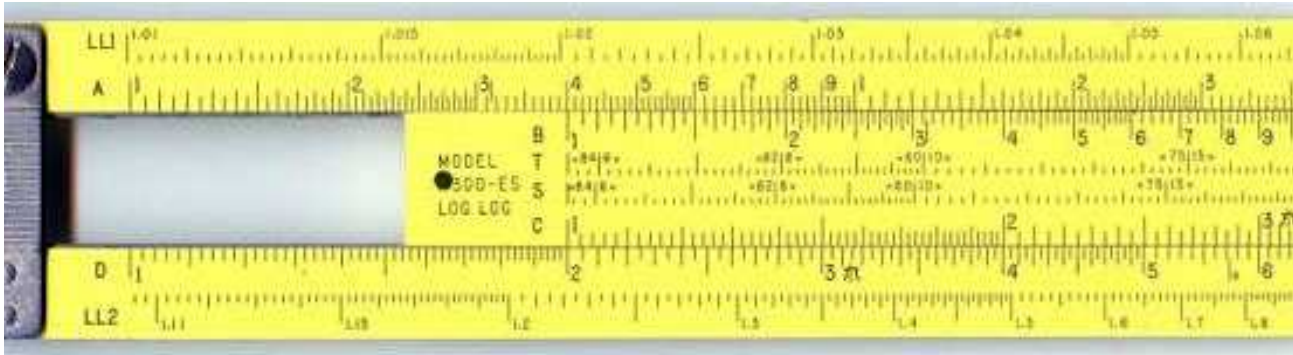
Exponents to base 10 are dropped, added mentally and reapplied afterwards. Mantissas multiplied using logarithmic addition.

A pair of logarithmic scales, one displaced linearly from the other by $\log X$.

Reading projected off from the second at offset $\log Y$ back to the first will be $\log^{-1}(\log X + \log Y)$.

Physical relative accuracy (say) 0.1 mm in 300 mm.

Bels and Decibles

An order of magnitude ($\times 10$) increase in power is one **bel**.

A tenth of a bel is a more useful unit - the decibel or dB.

One decibel is approximately the limit of sensory resolution (certainly for hearing).

The voltage from a microphone will increase relatively by $\sqrt{10^{1/10}}$ for a 1 dB increase in ambient noise power.

Human ear (safe!) dynamic range: 20×10^{-6} .. 20×10^{6} Pascals. A range of 120 dB (hard to match in any man-made analogue system).

HiFi/rock concert levels approaching 10^5 Pa (one atmosphere) require negative air pressure!

Using Decibels for Composed System Transfer Function Product
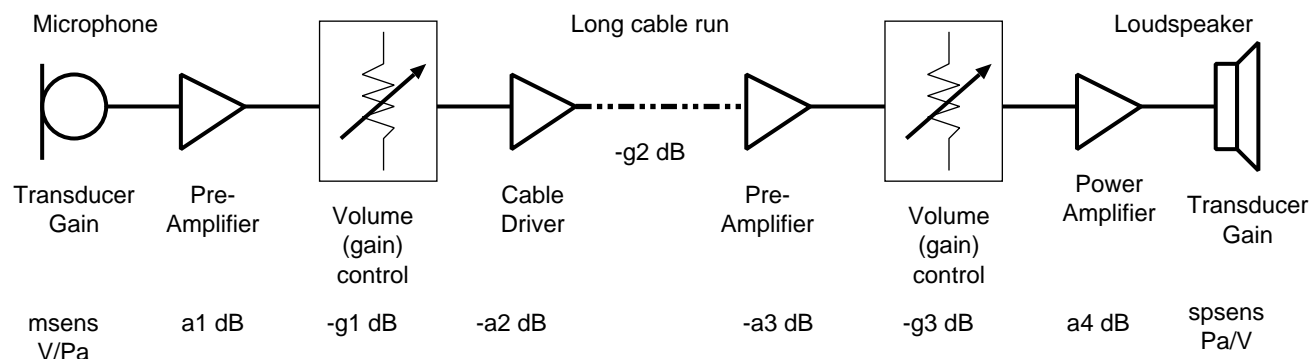
The overall gain of a chain of transducers, amplifiers and attenuators in a system is the product:

$$\frac{\text{output}}{\text{input}} = \text{msens} \times \prod g_i \times \prod a_j \times \text{spsens}$$

System gain is generally computed by summming component gain in decibels.

Amplifiers have positive gain in decibles. Attenuators have negative gain in decibels.

Requantising with Loss of Precision

y = floor(x)

Integer quantisation function
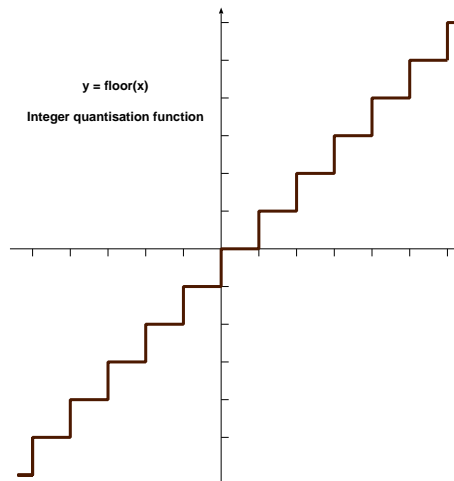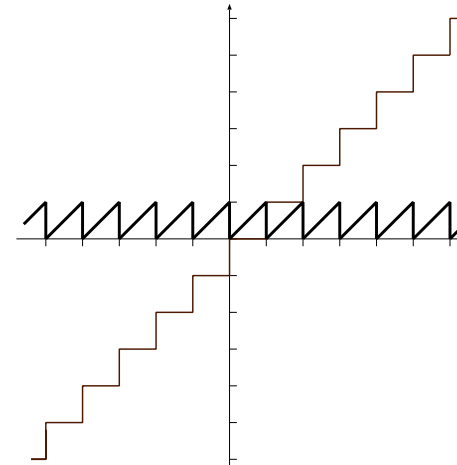
$$y = \mathsf{floor}(x) = \lfloor x \rfloor \qquad\qquad y = x - \lfloor x \rfloor$$

Math: Round to nearest integer: $$f_{\mathrm{quantise0}}(x) = \lfloor x + 0.5 \rfloor$$	Fixed: Round to multiple of 10^5 $$f_{\mathrm{q1E5}}(x) = (x/10^5) * 10^5$$ Does not even need the floor operator!	Floating: Discard 5 decimal digits when $x \approx 1.0$ $$f_{\mathrm{q5}}(x) = (x + 10^5) - 10^5$$

Second two quantisers do not require a special operator such as floor!

Digital Audio Quantisation Noise

CD-quality stereo sound uses the following parameters:

- Number of channels per frame: 2

- Sample rate: 44100 sps

- Sample precision: 16 bits linear, two's complement.

Any audio details smaller than 2^{-65536} of full scale are discarded. The discarded details are nominally replaced with an added noise that exactly cancels those details.

Best possible StoN ratio is $20 \log_{10}(2^{-65536}) = -96$ dB

NiCAM digital audio: 32 samples of 14 bit reduced to 10 and a 3-bit exponent added.
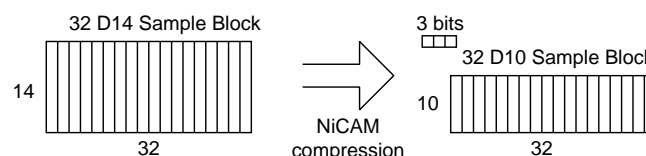
Near instantaneous compression and expansion: constant *relative* quantisation error. Subjectively lossless? *Exercise:* What is the StoN for NiCAM?

Non-uniform quanitsation: A-law Telephony Codec

G.711 A-law Telephony: 64 kbps 4kHz bandwidth digital audio
is used for all European landline calls:

- Number of channels per frame: 1

- Sample rate: 8000 sps

- Sample precision: 12+ bits mapped down to 8 using the A-law table.

A-law defined as a piecewise approximation to

$$y = Ax/(1 + \ln A) \quad \text{for } 0 \le x \le 1/A$$
$$y = 1 + \ln(Ax)/(1 + lnA) \quad \text{for } 1/A \le x \le 1$$
$$A = 87.6.$$
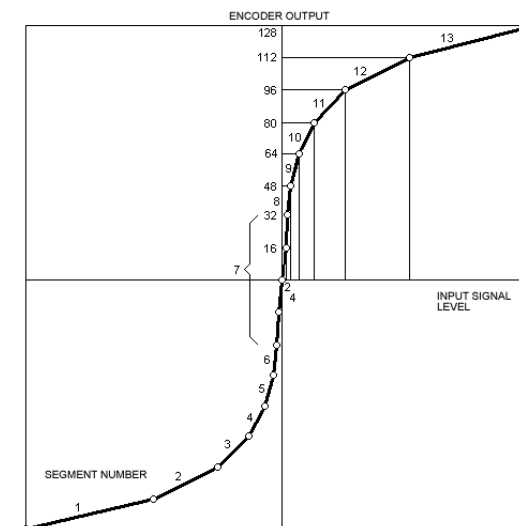
Again: roughly constant signal to quantisation ratio despite speech-varying signal level.

Exercise: What is the complexity of implementing A-law?   Details non-examinable.

Further Non-linear Encoding

IEEE Floating point stores a linear mantissa and a shorter, logarithmic exponent.

Certain algorithms (Viterbi and Baysian Inference) need to multiply all the time: preferred implementation may have zero length mantissa - just sign and exponent.

Custom encodings are typically used in specialist applications, such as: log-likelihood encoding in a low-density parity checker or in a neuron/brain simulator.

Binary	Meaning
S000	0
S001	1
S010	2
S011	3
S100	5
S101	10
S110	20
S111	inf

Matrix multiplication is generally well behaved.

$$\mathsf{R} = \mathsf{A}\,\mathsf{B} \text{ where } r_{ij} = \sum_k a_{ik} b_{kj}$$

See exercise sheet.

Over many centuries, integer multiplication was commonly done by lookup in table of squares, an addition and two subtractions.

$$a \times b \;\; \equiv \;\; \frac{(a+b)^2 - (a-b)^2}{4}$$

This is completely accurate provided $a + b$ is in the table.

The final division by four is trivial on binary computers but was factored into the printed tables when done by hand.

See exercise sheet.

Finite Differences Example 1 - Newtons's Cradle

Simulation: The centre balls stay still and we can even get an audio output from the collisions.

http://science.howstuffworks.com/newtons-cradle.htm
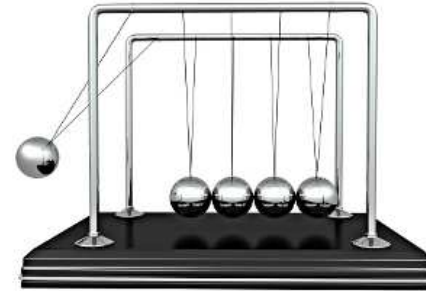
Assume strings inextensible and frame rigid. Rotation is constrained so the cradle becomes a 1-D linear system.

1. Ball: Treat as 1-D cylinder: need density and elastic modulus and linear elemental length.

2. Collisions: Treat initially as totally elastic (no energy loss).

3. Gravity: Assume restoring force acts equally on each elemental cylinder proportional to $\Delta m \sin \Theta$.

Try the files yourself for some number of balls: see TBA

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.

Try the files yourself and make a tune: see TBA

Nodal Analysis: solve the set of flow equations to find nodal potentials.
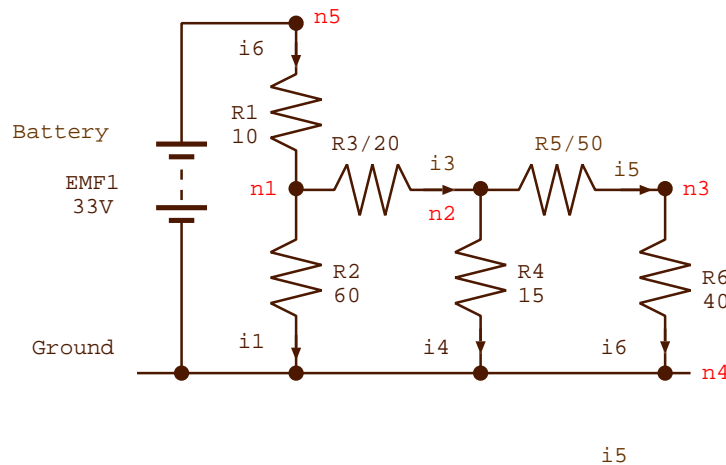
Electrical circuits generally have some voltage sources in them.

But our basic approach handles current sources only.
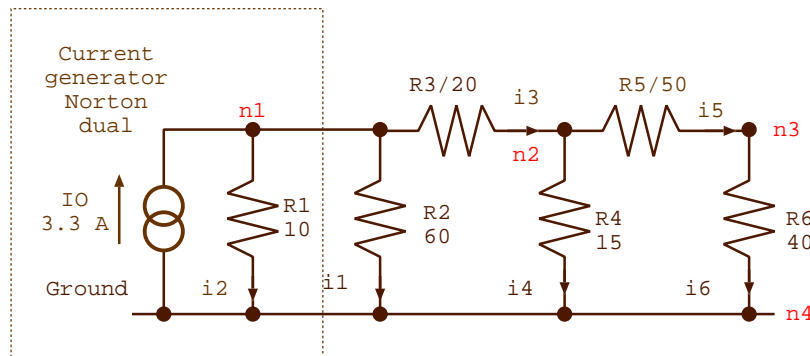
We will solve for the lower figure which is the Norton Dual (*non-examinable*) of the upper.

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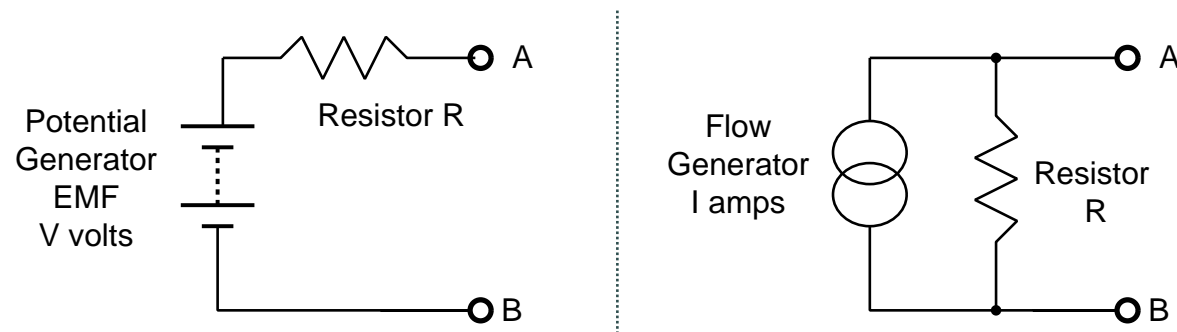
Provided $V = IR$, both have the same current at and potential between their terminals when connected to:

1. infinite resistance (open circuit),

2. infinite conductance (short circuit),

3. and under all other load conditions.

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.

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

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

$$-3.3 + i_2 + i_1 + i_3 = 0$$

$$-i_3 + i_4 + i_5 = 0$$

$$-i_5 + i_6 = 0$$

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

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

Note 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. The system is extended with voltage equations.

Semiconductors, like non-return valves, are non-linear.

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

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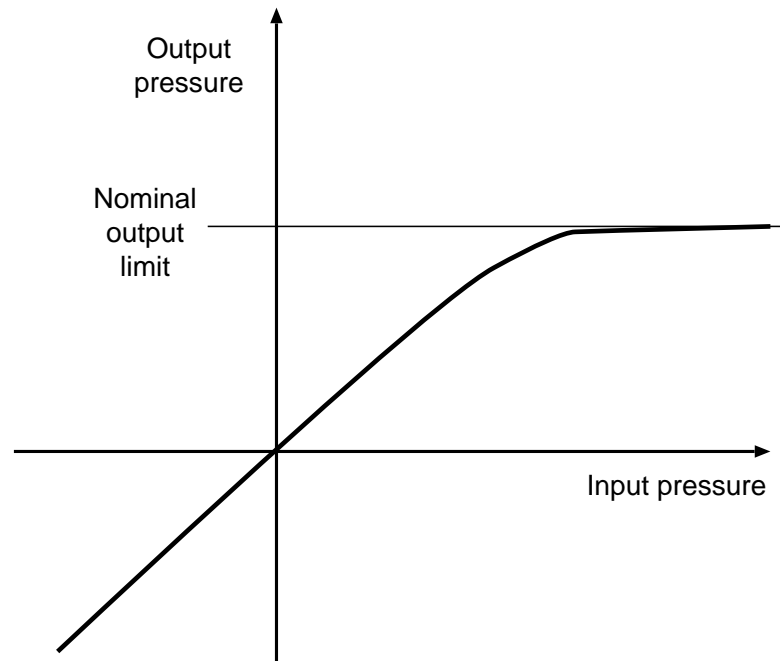$$

The diode's resistance changes according to its voltage! But a well-behaved derivative is readily available.
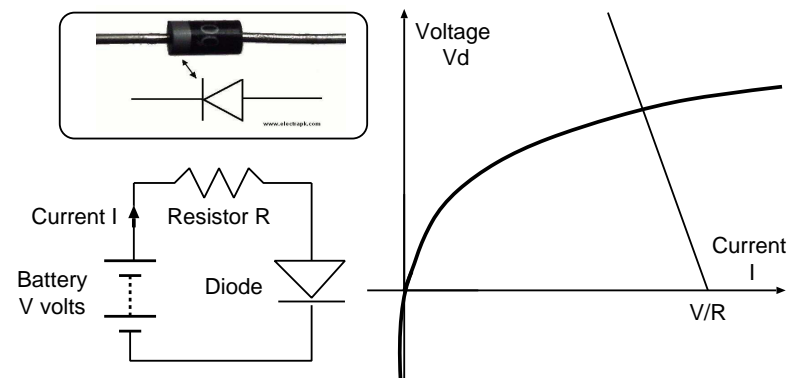
A non-linear component's tangent does not pass through the origin. An offset must be added.

Pressure regulator valves have a strongly non-linear behaviour!

The resistance increases drastically above the target output pressure.

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.

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.

$$
\begin{aligned}
G_d &= dI/dV = \frac{I_S}{nV_T} e^{V_d/(nV_T)} \\
I_d &= V_d G_d
\end{aligned}
$$
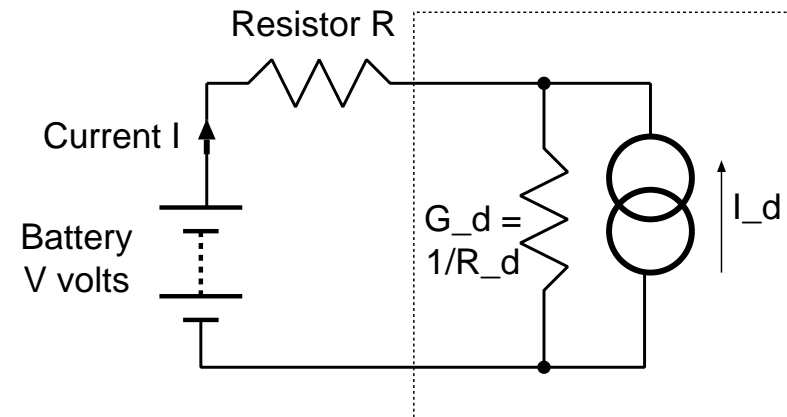
When we solve the circuit equations for a starting (G_d, I_d) pair we get a new V_d. We must iterate.

Owing to the differentiable, 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.
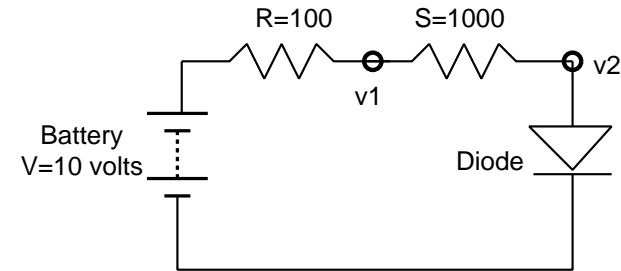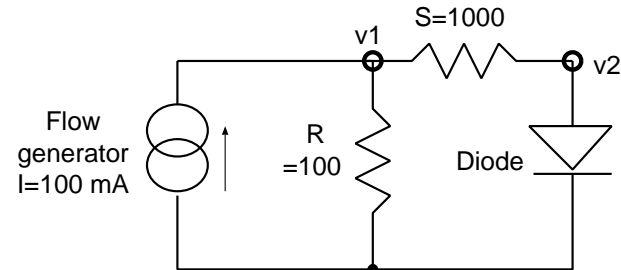
$$\sigma_{\text{fudge}} = 1/(0.2 \text{ ohms})$$

$$G_d = dI/dV = \frac{10^{-12}}{0.026} e^{V_d/0.026} + \sigma_{\text{fudge}}$$

$$I_d = V_d G_d$$

$$\begin{pmatrix} 1/100 + 1/1000 & -1/1000 \\ -1/1000 & 1/1000 + G_d \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0.1 \\ I_d \end{pmatrix}$$

Now, G_d is a function of $V_d = v_2$ so need to iterate.
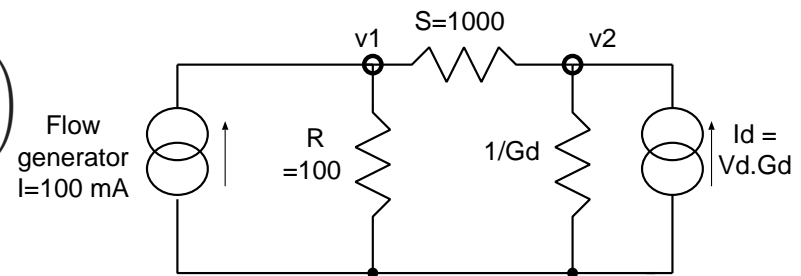
Note: I_d is not the current in the diode. It is the offset current to model the non-linearity.

The small fudge resistance typically has to be added to diodes to aid convergence.

Smart implementations iterate with multi-dimensional Newton Raphson.

```
javac DiodeExample2.java
java DiodeExample2
Diode2 iteration=0    v_1=0.000000 v_2=0.000000  (I_d=0.000000e+00) I=0.000000e+00
Diode2 iteration=200   v_1=9.107288 v_2=0.180167  (I_d=1.801672e+00) I=8.927121e-03
Diode2 iteration=400   v_1=9.123372 v_2=0.357087  (I_d=3.571000e+00) I=8.766284e-03
Diode2 iteration=600   v_1=9.139102 v_2=0.530119  (I_d=5.447185e+00) I=8.608982e-03
Diode2 iteration=800   v_1=9.149486 v_2=0.644341  (I_d=2.079746e+01) I=8.505145e-03
Diode2 iteration=1000   v_1=9.152371 v_2=0.676083  (I_d=5.782025e+01) I=8.476288e-03
Diode2 iteration=1200   v_1=9.153743 v_2=0.691175  (I_d=1.001847e+02) I=8.462568e-03
Diode2 iteration=1400   v_1=9.154629 v_2=0.700921  (I_d=1.446125e+02) I=8.453708e-03
Diode2 iteration=1600   v_1=9.155281 v_2=0.708086  (I_d=1.901970e+02) I=8.447194e-03
Diode2 iteration=1800   v_1=9.155795 v_2=0.713740  (I_d=2.365510e+02) I=8.442055e-03
Diode2 iteration=2000   v_1=9.156219 v_2=0.718404  (I_d=2.834687e+02) I=8.437814e-03
Diode2 iteration=2200   v_1=9.156579 v_2=0.722371  (I_d=3.308253e+02) I=8.434208e-03
Diode2 iteration=2400   v_1=9.156893 v_2=0.725821  (I_d=3.785387e+02) I=8.431072e-03
Diode2 iteration=2600   v_1=9.157170 v_2=0.728872  (I_d=4.265507e+02) I=8.428298e-03
Diode2 iteration=2800   v_1=9.157419 v_2=0.731606  (I_d=4.748189e+02) I=8.425813e-03
Diode2 iteration=3000   v_1=9.157644 v_2=0.734083  (I_d=5.233106e+02) I=8.423561e-03
Diode2 iteration=3200   v_1=9.157850 v_2=0.736346  (I_d=5.720001e+02) I=8.421503e-03
Diode2 iteration=3400   v_1=9.158039 v_2=0.738430  (I_d=6.208671e+02) I=8.419609e-03
Diode2 iteration=3600   v_1=9.158215 v_2=0.740360  (I_d=6.698945e+02) I=8.417854e-03
Diode2 iteration=3800   v_1=9.158378 v_2=0.742158  (I_d=7.190683e+02) I=8.416220e-03
Diode2 iteration=4000   v_1=9.158531 v_2=0.743841  (I_d=7.683767e+02) I=8.414690e-03
Diode2 iteration=4200   v_1=9.158675 v_2=0.745421  (I_d=8.178093e+02) I=8.413253e-03
Diode2 iteration=4400   v_1=9.158810 v_2=0.746912  (I_d=8.673576e+02) I=8.411898e-03
Diode2 iteration=4600   v_1=9.158938 v_2=0.748322  (I_d=9.170137e+02) I=8.410616e-03
Diode2 iteration=4800   v_1=9.159060 v_2=0.749660  (I_d=9.667710e+02) I=8.409400e-03
```

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

Water flows down a pipe at a rate proportional to entry pressure.

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

$$\mathrm{d}V_c/\mathrm{d}t = -I/C$$
$$I = V_c G$$

$$\mathrm{d}d/\mathrm{d}t = -f/A$$
$$f = \rho d g/kl$$
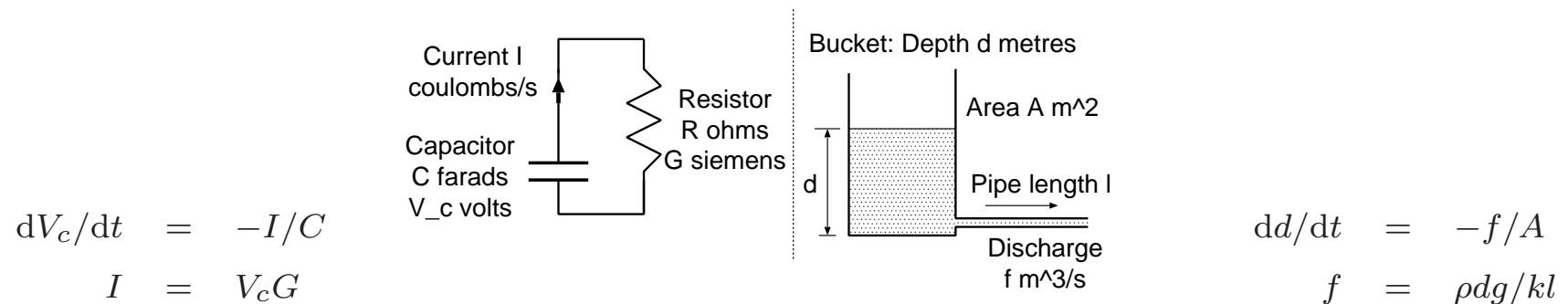
At any timepoint, the capacitor/bucket has a fill level and a flow rate: and hence a 'conductance' ratio between them.

Perform time-domain simulation with a suitable ΔT.

These time-varying components are linearised, but other non-linear components (like the diode) are likely to be present and so will require a Newton iteration in each timestep.

Non-examinable: 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.

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

Capacitor/bucket
replacement

Existing
remainder of
the circuit...
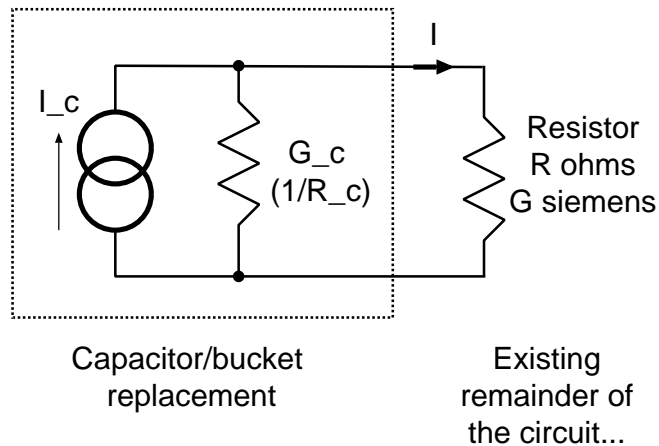
$$
\frac{dV}{dI} = \frac{d(\Delta V)}{dI} = \frac{d(I\Delta T/C)}{dI} = \frac{\Delta T}{C}
$$

$$
G_c = \frac{dI}{dV} = \frac{C}{\Delta T}
$$

$$
I_c = V_c G_c
$$

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

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

Let's use 10 μsecond timestep: $\Delta T = 10^{-5}$.

We need an initial capacitor charge.
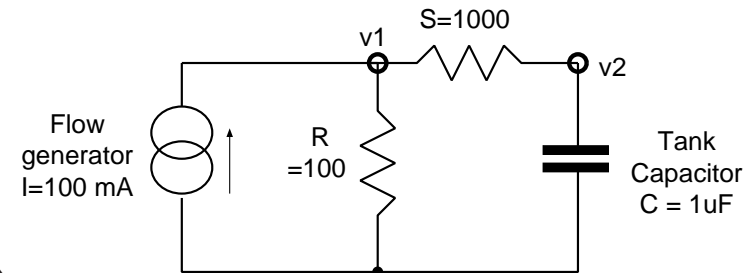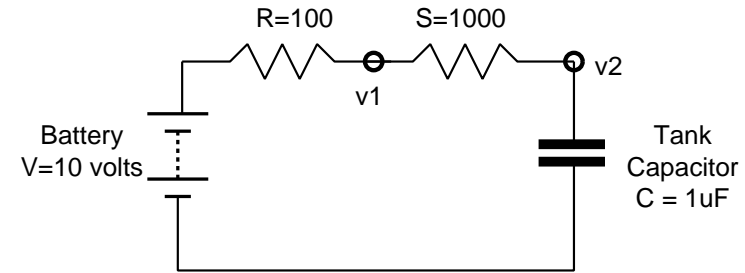E.g. $V_c^{(0)} = 0.0$V.

$$V_c = v_2.$$
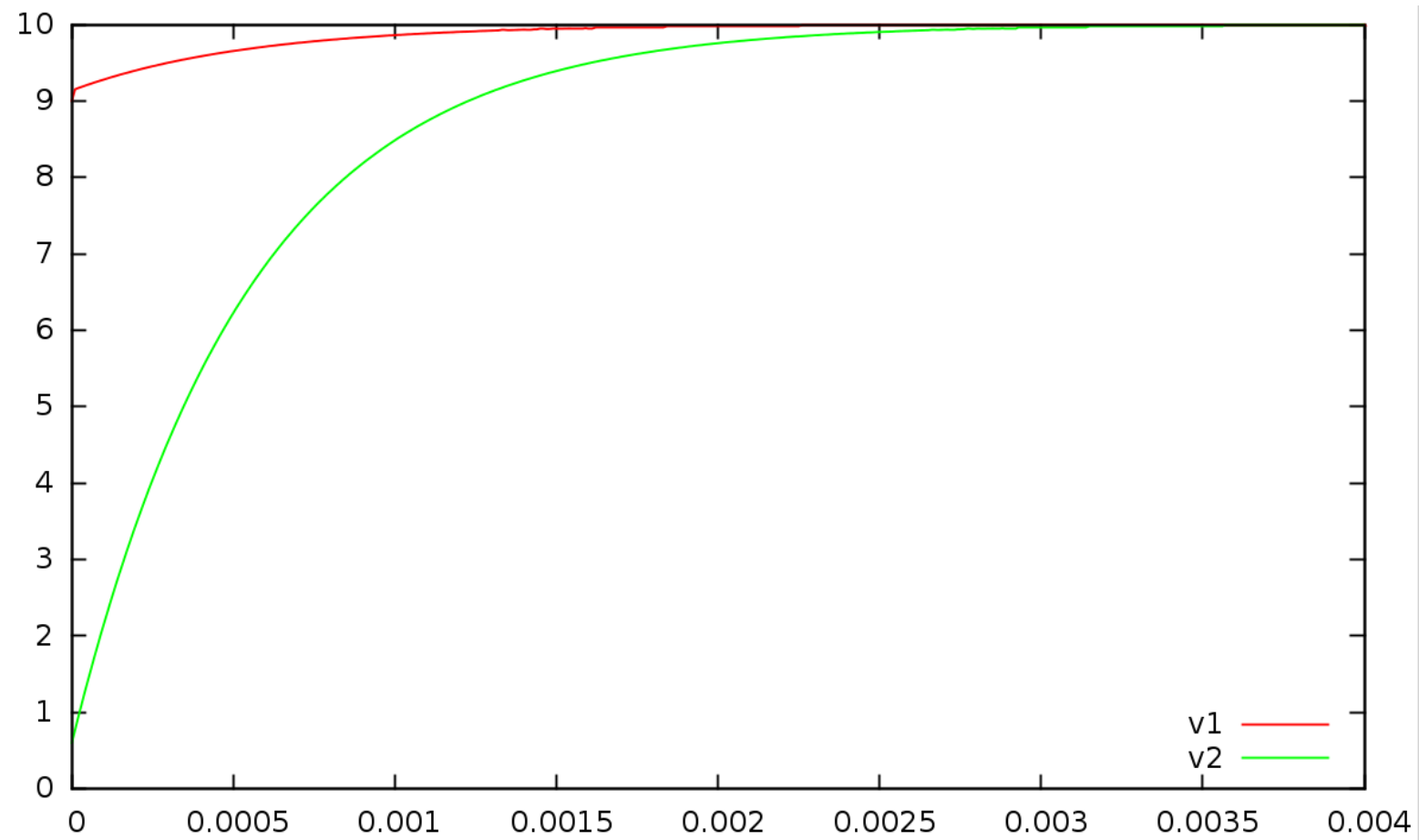
$$G_c = dI/dV = C/\Delta T = 10.0^{-6}/10^{-3}$$

$$I_c = V_c G_c$$

$$\begin{pmatrix} 1/100 + 1/1000 & -1/1000 \\ -1/1000 & 1/1000 + G_c \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0.1 \\ I_c \end{pmatrix}$$

Now, G_c will change with time. Need to iterate over time.

$$V_c^{(t+\Delta T)} = V_c^{(t)} - \frac{I\Delta T}{C}$$
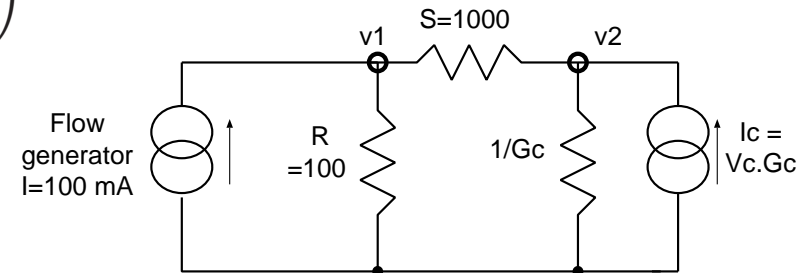
Generated by the file JPrograms/CapacitorExample2.java

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 (linear forward differences) (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.

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 $N_{\mathrm{it}} < N_{\min}$ { $\Delta T* = 1.1$;}

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.
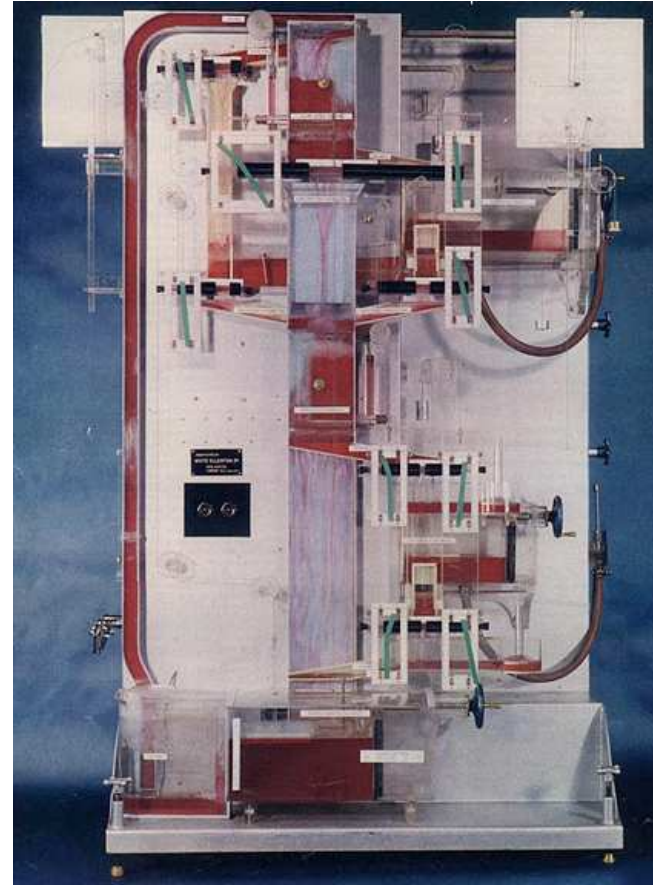
Exercise: Please see exercise sheet.

What is the expected accuracy when averaging N points collected from simulation?

> Variance of N i.i.d. RVs is $N\sigma^2$.
>
> Standard deviation of measured statistic is $\sqrt{N\sigma^2}/N = \sigma/\sqrt{N}$

So averaging is a well-behaved, if somewhat slow, convergence.

___

What about order statistics like the 99th percentile?

> The n^{th}th percentile of an observation variable is the value that cuts off the first n percent of the data values when it is sorted in ascending order.

Only 1 percent of data fall in the 99th-tile so convergence even slow.

Solution: use *batching* aka *sectioning* or even *Jackknifing*.

Non-examinable.   *Exercise for the keen:* See exercise sheet

If we have a multivariate function where we know the derivative we can do simple **gradient decent** by repeatedly stepping backwards along the derivative vector.

Earlier I glossed over how to do iterations with matrices.

If we know the partial derivative of each output variable y^T for each input variable x^T we can use Newton's iteration to find x such that $Ax = y = 0$.

A matrix of partial derivatives of a matrix with respect to a column vector is known as the **Jacobian** of that matrix.

We need to invert the Jacobian owing to Newton needing to divide by the derivative ...

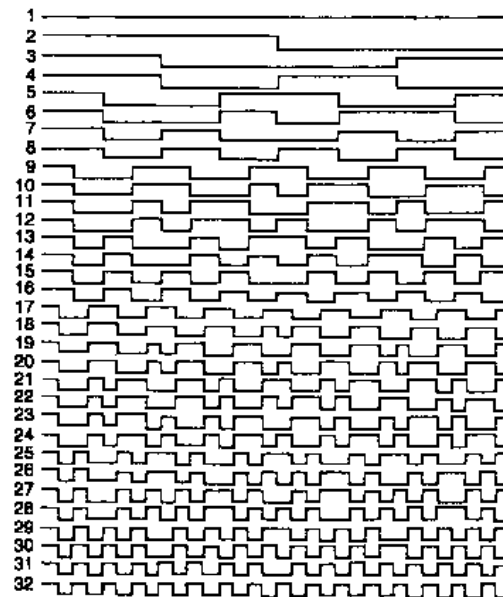*This slide is non-examinable and may be lectured if time permits.*

This slide is non-examinable.

When we do not have the system derivatives we can estimate them using small perturbations of the inputs and then do a larger Newton step.

An orthogonal set of small perturbations is needed of proportional to the the number of input variables:

1. Use small unit steps?

2. Use the Walsh Functions.

Walsh Functions: an orthogonal basis set of binary functions.

There are many optimisation packages available.

They can be used as controlling wrappers around your own system model or simulation.

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

- MATLAB can solve linear, quadratic, and nonlinear problems with Optimization Toolbox

- OpenOffice has a plugin for its spreadsheet that performs non-linear regression.

Other packages can do useful jobs like single-value decomposition (SVD).

This slide is non-examinable.

The End.

Please do the online exercises and study the tiny example programs.

Thankyou and goodbye.

(C) DJG April 2014.