2018

Numerical Methods Notes

UNIVERSITY OF CAMBRIDGE, PART IA

ASHWIN AHUJA

Table of Contents

Numerical Methods

Part 1: Reminders

We must always be careful about when a computer program is working – we should implement unit tests to check if the program is correct. We must also be aware that even if a program 'implements' a mathematical formula, they are not necessarily correct.

What causes errors:

1. Wrong mathematical model
2. Overfitting – parameters chosen to fit the expected value
3. Model being too sensitive to input values
4. Discretisation of continuous model
5. Propagation of inaccuracies (floating point inaccuracies)
6. Programming errors

Integer Representation

Signed and Unsigned integers are two methods of representation (with signed having a bit at the front which says whether the integer is positive or negative). These place the decimal point at the end.

We can create a **fixed-point number** by placing the decimal point somewhere else in the number. These however, are prone to overflow. We can use **saturating,** therefore 2 * 10000111 = 11111111 (therefore goes to highest representable value). However, we can much more easily reduce the overflow by allowing the decimal point to be determined at run-time – this is **floating point** numbers

Scientific Notation

$a \times 10^b$. a is any real numbers, called the significand or mantissa. B is the exponent. In normalised form, $1 \leq a < 10$

Multiplication and Division: $x_0 = a_0 \times 10^{b0}$, $x_1 = a_1 \times 10^{b1}$

$x_0 \times x_1 = (a_0 \times a_1) \times 10^{b0 + b1}$

$x_0 / x_1 = (a_0 / a_1) \times 10^{b0 - b1}$

Addition and Subtraction: Requires the numbers to be represented using the same exponent – normally the larger of b0 and b1

Say b0 > b1 => $x_1 = (a_1 \times 10^{b1 - b0}) \times 10^{b0}$

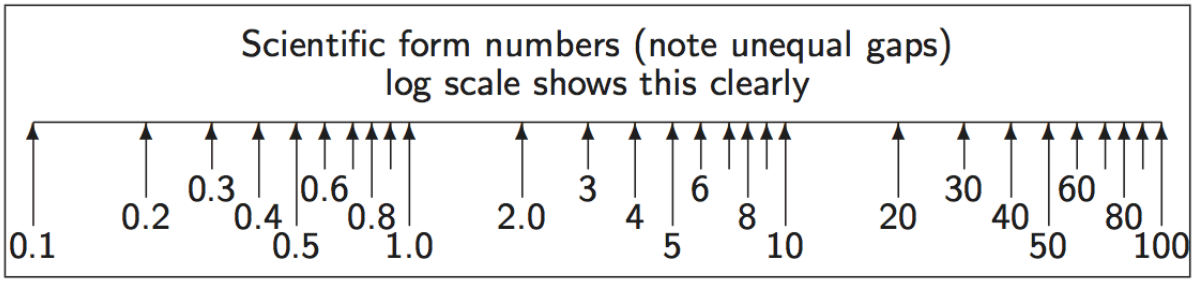
$x_0 \pm x_1 = (a_0 \pm (a_1 \times 10^{b1 - b0})) \times 10^{b0}$

This may require a number of shifts to normalise the result.

Significant Figures

While significant figures are much used, they are generally a little risky:

1. There may be 3sf in the result of a computation, but little accuracy left
2. Changing the ulp (unit in last place) may change the value by different amounts – **relative error**



Computer Representation

Given a number represented as β^e x d0.d1…d_{p-1} we call β the base (or radix) and p the precision. Since e has a fixed-width finite encoding, its range must also be specified. All contemporary, general purpose, digital machines use binary and keep β = 2.

Long Multiplication: Broadside Addition

Multiplying or dividing by powers of 2 uses shifts which are much cheaper in binary hardware.

```
fun mpx1 (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 mpx1 (x', y', c', carry')
        end;
```

Long Division

Variable Latency (Standard Long Division)

```
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 ship loop *)

        fun mainloop N D p r =
            if p=0 then r
            else
                (* Binary Decision, either goes up or doesn't *)
                let val (N, r) = if N >= D then (N-D, r+p) else (N, r)
            in mainloop N (D div 2) (p div 2) r
            end
    in mainloop N D p 0
    end;
```

Fixed-Latency Long Division

```
val NUMBASE = 1073741824 (* 2^30 *)
fun divide N D =
    let fun divloop (Nh, N1) p q =
        if p=0 then q
        else
            let val (Nh, N1) = (Nh*2 + N1 div NUMBASE, (N1 mod NUMBASE)*2)
```

```
                val ((Nh,N1), q) = if Nh >= D then ((Nh - D, N1), q+p) else ((Nh,
N1), q)
            in divloop (Nh, N1) (p div 2) q
            end
    in divloop (0, N) NUMBASE 0
    end;
```

Integer Input (ASCII to Binary)

```
fun asciiToInteger [] c = c
|   asciiToInteger (x :: xs) c = asciiToInteger xs (c*10 + ord x - ord #"0");
```

Integer Output (Binary to ASCII)

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

Floating Point Output (Double Precision to ASCII)

This depends on the number of significant figure, say 4. In order to do this, we scale the number to between 1000 and 9999. We then cast it to an integer, converting the integer to ASCII as normal, but either insert a decimal point, or add a trailing exponent denotation.

```
val new_tens_table = Vector.fromList [1E0, 1E1, 1E2, 1E3, 1E4, 1E5, 1E6, 1E7, 1E8];
val binaryExponentsTable = [(1E32, 32), (1E16, 16), (1E8, 8), (1E4, 4), (1E2, 2),
(1E1, 1)];
val binaryFractionsTable = [(1E~32, 32), (1E~16, 16), (1E~8, 8), (1E~4, 4), (1E~2,
2), (1E~1, 1)];

fun floatToString precision d00 =
    let val lowerBound = Vector.sub(new_tens_table, precision)
        val upperBound = Vector.sub(new_tens_table, precision + 1)
        val (d0, sign) = if d00 < 0 then (0-d00, [#"-"]) else (d00, [])
        fun chop_upwards ((ratio, bitpos), (d0, exponent)) =
            if (d0 * ratio) < upperBound then ((d0 * ratio), exponent - bitpos) else
(d0, exponent)
        fun chop_downwards ((ratio, bitpos), (d0, exponent)) =
            if (d0 * ratio) > lowerBound then ((d0 * ratio), exponent + bitpos) else
(d0, exponent)
        val (d0, exponent) =
            if (d0 < lowerBound) then foldl chop_upwards (d0, 0) binaryExponentsTable
```

```
            else foldl chop_downwards (d0, 0) binaryFractionsTable
        val imant = floor d0 (* convert mantissa to integer *)
        val exponent = exponent + precision
        (* Decimal point will only move a certain distance: outside that range, we
force scientific form *)
        val scientific_form = exponent > precision orelse exponent < 0
        fun digits d0 p trailZeroSuppression =
            if p0<0 orelse (trailZeroSuppression andalso d0 = 0) then []
            else
                let val d = d0 div Array.sub(new_tens_table, p)
                    val r = d0 - d * Array.sub(new_tens_table, p)
                    val dot_time = (p= precision + (if scientific_form then 0 else
0-exponent))
                    val rest = digits r (p-1) (trailZeroSuppression orelse dot_time)
                    val rest = if dot_time then (#"."::rest) else rest
                in if d>9 then #"?" :: binaryToAscii d0 @ #"!" :: rest else
chr(ord(#"0") + d) :: rest
                end;
        val mantissa = digits imant precision false
        val exponent = if scientific_form then #"e" :: binaryToAscii exponent else
[]
    in (d00, imant, implode (sign @ mantissa @ exponent))
    end;
```

Ceiling and Floor Functions

Ceiling: Nearest integer rounding up
Floor: Nearest integer rounding down

Rounding Fairly

```
int round (double arg)
{
    return Floor(arg + 0.5);
}
```

Part 2: Floating Point Representation

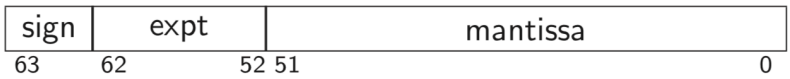
Standards

In the past, every manufacturer produced their own floating-point hardware and floating point programs gave difficult answers. The IEEE standardisation fixed this:

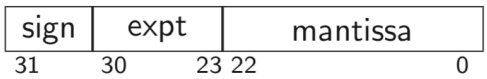
- Two different IEEE standards for floating-point computation
- IEEE 754 – β = 2, p (number of mantissa bits) = 24 for single precision or p = 53 for double precision
- Later augmented to include longer binary floating point formats and also decimal floating formats
- IEEE 854 is more general and allows binary and decimal representation without fixing the bit-level format

IEEE 754

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



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



History

- Every computer had its own floating-point format with its own form of rounding – so floating-point results differed from machine to machine.
- Despite mumblings, the IEEE standard largely solved this – therefore has stood the test of time.
- Intel had the first implementation of IEEE 754 in its 8087-co-processor chip to 8086 (and drove lots 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 (preferred speed over accuracy)
- The SSE2 Instruction Set, available since Pentium and still used, includes a separate instruction set which better enables compiler writers to generate fast (and IEEE-valid) floating-point code.

Hidden Bit and Exponent Representation

The advantage of the base-2 exponent is representation is that all normalised numbers start with a 1, therefore we do not need to store it. However, to deal with 0, we remove the highest and lowest exponent values and make them represent 0 and infinity respectively.

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)

This representation is called excess-127 (127 bias) for single precision floats or excess-1023 for double precision.

Division by a Constant

In order to divide by ten, you multiply by the reciprocal using the following code:

```
// In binary 1/10 = 0.0001100110011001...
```

```
unsigned div10 (unsigned int n)
{
    unsigned int q;
    q = (n >> 1) + (n >> 2);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    return q >> 3;
}
```

Signed Zeros

Signed Zeros, while seemingly useless, are in fact sometimes useful. For example, if a series of operations lead to an underflow, it is useful to know that it came from positive or negative. Still can't do arithmetic with them – eg. 1 / -0 = -inf

Exceptions

Overflow Exception: Using floating point, occurs when an exponent is too large to be stored. Occur mostly through divisions and multiplications, though also through division by zero.

Underflow Exception: When the result is too low. Generally, no exception is thrown, it is just left alone to that result.

Exceptions vs Infinities vs NaNs

- Alternatives to infinities and NaNs are to give a wrong value or to throw an exception
- An infinity (or a NaN) propagates through a calculation
- Raising an exception is likely to cause the computation to fail and giving wrong values is dangerous

Part 3: Floating Point Operations

IEEE Arithmetic

IEEE Basic Operations (+, -, *, /) are defined as follows:

- Treat the operations as precise, do perfect mathematical operations on them. Round this mathematical value to the nearest representable IEEE number and store this as a result. In the event of a tie, choose the value with the even (zero), least significant bit.

IEEE Rounding

IEEE requires there to be a global flag to be set to be one of 4 values – which says how the rounding occurred.

- **Unbiased**
  - 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 is required to be default
- **Towards zero**
- **Towards positive infinity**
- **Towards negative infinity**
  - These all introduce a systematic bias

Other Operators

- Other mathematical operators are typically implemented in libraries – these examples include sin, sqrt, log. It is important to ask whether implementations of these satisfy the IEEE requirements.
- Generally, a library is only as good as the vendor's careful explanation of what error bound the result is accurate to.
- ±1 ulp is considered good – 0.5 ulp is required for a perfect accuracy result.
- The results must also be **semi-monotonic**
    - Given the function is monotonically increasing, the libraries answers must also be monotonically increasing.

Errors

Quantisation Errors: Arising from the inexact representation of constants in the program and numbers read in as data.
Rounding Errors: Produced by every IEEE operation
Truncation Errors and also underflow and cancellation / loss of significance.

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

Errors in +, -: these sum the absolute errors
Errors in *, /: these sum the relative errors

Error amplification is generally far more important than the random error through floating point computation. (Systematic Error Propagation)

Machine Epsilon

- Defined as the difference between 1.0 and the smallest representable number which is greater than 1 (2^{-23} in single precision)
- Useful as it gives an upper bound on the relative error caused by getting a floating-point number wrong by 1 ulp and is therefore useful for expressing errors independent of floating point size.
- Wrong description
    - Smallest number which when added to one gives a number greater than 1
    - With rounding to nearest, this only needs to be slightly more than half of our machine epsilon
    - Error used by Microsoft and even the GNU documentation
    - It is almost the maximum error but not quite
- **Negative Epsilon**
    - Difference between 1 and the smallest representable number greater than 1
    - This is exactly 50% of machine epsilon
        - Witching-Hour effect
        - 2^0 x 1.0000…

    - Closest number above is 2^0 x 1.000000…1
    - Closest number below is 2^{-1} x 1.111…1
    - Therefore, this ulp represents only half as much as the ulp in closest number above.
- Even Machine Epsilon can build up over time to create a large error.

Part 4: Simple Maths, Simple Programs

Map-Reduce

An example is:

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

The work in parallel steps is independent and results are only combined at the end under a commutative and associative operator.

When finding a root of an equation, we have a totally different type of iteration:

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

Iterative processes need techniques, since the program may be locally sensible, but a small representation or rounding error can slowly grow over many iterations.

Ensuring relative error is small

1. For example, for quadratic equation (smaller root in magnitude)
2. $\frac{-b+\sqrt{b^2-4ac}}{2a}$
3. $= -\frac{2c}{b+\sqrt{b^2-4ac}}$
4. The second expression computes the root much more accurately (for the smaller root)

Summing a Finite Series

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

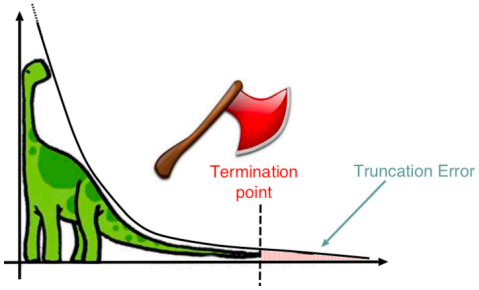
- This clearly sums to infinity – divergent series, however, if we calculate it using a float until it stops growing – it = 13.849. If we do it in reverse, we get the same value for a double precision floating point number, but don't for single precision floating point values.
- If we add a + b + c, it is generally more accurate to sum the smaller values and then add the third.
- Therefore, it is better to sum starting with the smallest number and then going towards the larger numbers.
- Also, there is an algorithm called Kahan's Summation Algorithm
    - Uses a second variable which approximates the error in the previous step which can then be used to compensate in the next step.

Part 5: Infinite and Limiting Computations

Different Types of Error

1. Rounding Error

   a. Error we get by using finite arithmetic during a computation
2. Truncation Error (Discretisation Error)
   a. Error we get by stopping an infinitary process after a finite point



Differentiation

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

How to pick h:

- We technically want it to be as small as possible
- But if h is less than the machine epsilon then for x about 1, x + h will compute to the same value as x
- Also, if h is too small, then f(x+h) – f(x) will produce lots of cancelling – therefore a lot of relative error – **rounding error**
  - Assume machine epsilon error is returned in evaluations of f and get (assume f(x) and f'(x) $\cong 1$
  - (f(x+h) – f(x))/h = (f(x) + hf'(x) ± macheps – f(x))/h
  - 1 ± macheps/h
  - Therefore rounding error is macheps / h
- However, if h is too big, we create **truncation error**.
  - Truncation error can be calculated by Taylor expansion
  - $f(x+h) = f(x) + hf'(x) + \frac{h^2 f''(x)}{2} + O(h^3)$
  - This works out as approximately $\frac{hf''(x)}{2}$
- Since errors vary oppositely with regards to h, we compromise by making the two errors of the same order to minimise their total effect.
- Equating the rounding error and the truncation error gives h = macheps/h
  - That is h = $\sqrt{macheps}$

However, can also do $\frac{f(x+h) - f(x-h)}{2h}$

- Truncation error = $h^2 f''(x)/3!$
- Rounding error = macheps/h
- Therefore h = $\sqrt[3]{macheps}$

When finitely approximating a limiting process, there is a number h which is small, or a number n which is large. Often, there are multiple algorithms which mathematically have the same limit, but approach at different speeds – as well as having different rounding error accumulation.

Order of an Algorithm

The way in which truncation error is affected by reducing h (or increasing n) is called the order of the algorithm wrt to that parameter. For example, (f(x+h) – f(x))/h is a first order method of approximating derivatives of smooth functions (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.

By having higher-order methods, we can use a relatively large h without incurring excessive truncation error (this reduces the rounding error too)

Root Finding: f(x) = 0.

Bisection Method:

- Form of successive approximation:
    1) Choose initial values a, b st sign(f(a)) != sign(f(b))
    2) Find midpoint c = (a+b)/2
    3) If |f(c)| < desired_accuracy then stop, otherwise
    4) If sign(f(c)) == sign(f(a)) a=c; else b=c
    5) goto line2
- The absolute error is halved at each step, so it has first-order convergence
    - **First-order convergence requires a number of steps proportional to the logarithm of the desired numerical precision**
- This is also known as a binary chop and it clearly gives one bit per iteration.

Convergence Iteration: Consider the golden ration ($\phi^2 = \phi + 1$)

With iteration $x_{n+1} = \sqrt{x_n + 1}$

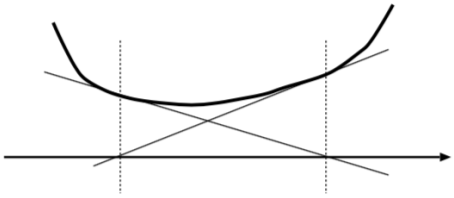
It converges, and error reduces by a constant fraction each iteration

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

(this is linear convergence)

But, with iteration $x_{n+1} = x_n^2 - 1$, the iteration diverges. It enters a **limit cycle**

Limit Cycle



1) Mathematically correct – function just fails to touch the x-axis
2) May arise from discrete and non-continuous floating-point details

Iteration Choices:
1) What iteration to use
2) When to stop the iteration (termination criterion)
    a. After number of cycle
    b. After two iterations result in same value
    c. When iteration 'moves' less than a given relative amount
    d. When error stops decreasing
    e. When error is within a pre-determined tolerance

Newton-Raphson
Given an equation of the form f(x) = 0, then the Newton-Raphson iteration improves an initial estimate x_0 of the root, by repeatedly setting:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

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

This is second-order (quadratic) convergence
- Quadratic convergence means that the number of accurate decimal (or binary) digits doubles with every iteration
- We have problems if |f'(x)| starts small
- There is the possibility of loops
- And behaves badly near multiple roots

Summing a Taylor Series
Taylor Series unconditionally converges everywhere, therefore many problems can be solved by summing a Taylor Series

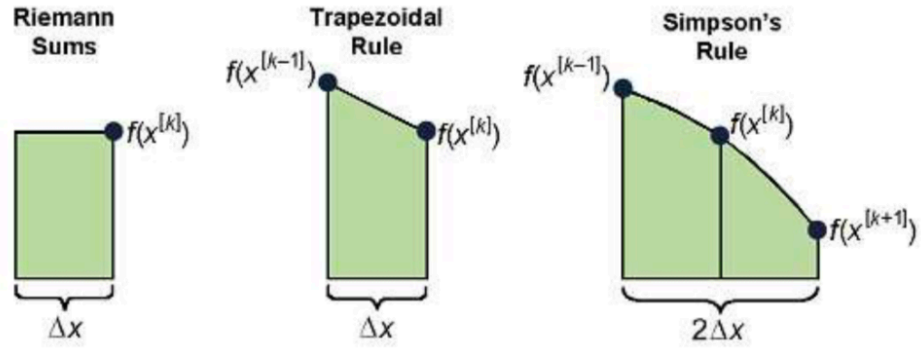
Issues:
1) How many terms to go to? (stopping early gives a truncation error)
2) Large cancelling intermediate terms can cause a loss of precision (rounding error)

Solutions for sin(x):
1) Do **range reduction –** use identities to reduce the argument to the range [0, pi/2]

a. But might need a lot of work to do this – since we need pi to a large accuracy

2) Now we can choose a fixed number of iterations and unroll the loop.

Definite Integrals Between Definite Limits



- Mid-point rule – puts a horizontal line at each ordinate to make rectangular strips
- Trapezium rule – uses an appropriate gradient line through each ordinate
- Simpson's rule – fits a quadratic at each ordinate

Quadrature: How many strips to use?

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

The rounding noise will be a random walk proportional to $\sqrt{n}$. The truncation noise depends on the nature of f(x) – 0 for quadratics and several classes of higher-order polynomial.

Lagrange Interpolation

Information Transform Theory tells us that we can fit an n^{th} degree polynomial through the n = k + 1 distinct points: (x_0, y_0), …, (x_j, y_j), …, (x_k, y_k)

Lagrange does this with the linear combinations:

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

$$l_j(x) = \prod_{0 \le m \le k}^{m \ne j} \frac{x - x_m}{x_j - x_m} = \frac{x - x_0}{x_j - x_0} \cdots \frac{x - x_{j-1}}{x_j - x_{j-1}} \frac{x - x_{j+1}}{x_j - x_{j+1}} \cdots \frac{x - x_k}{x_j - x_k}$$

The l_l are an orthogonal basis set: one being unity and the remainder zero at each datum. However, the errors can be large at the edges. Fourier clearly works well with a periodic sequence (that has no edges).

Splines and Knots

- **Splining**: Fitting a function to a region of data such that it smoothly joins up with the next region.
- Simpson's quadrature rule returns the area under a quadratic spline of the data
- If we start a new polynomial every k points, we will generally have a discontinuity in the first derivative upwards.

- *If we overlap spines, we get smoother joins*

Chebyshev Polynomials

Using Chebyshev Polynomials, we achieve Power Series Economy – total error in a Taylor expansion is re-distributed from the edges of the input range to throughout the input range.

Chebyshev Polynomials are simply a small adjustment of Taylor's values. The computations then proceed identically.

Volder's Algorithm – CORDIC

CORDIC: Coordinate Rotation Digital Computer

Allows us to find things like sine and cosine of Θ by rotation the unit vector (1, 0)

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

$$\Theta = \sum_i \alpha_i$$

Effectively, we subdivide the rotation into a composition of easy-to-multiply rotations until we reach the desired position.

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

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

We can then use a precomputed table containing decreasing values of a_i st each rotation matrix contains only negative powers of 2. 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 ≅ 0.5, tan 14.036 ≅ 0.25

We handle the scalar coefficients using a proper multiply with lookup value from a second, precomputed table.

$$scales[n] = \prod_{i=0}^{n} \frac{1}{\sqrt{1+\tan^2 a_i}}$$

```
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);
}
```

Accuracy: Very hard to reach the value of IEEE basic operations (result must be the nearest IEEE representable number to the mathematical result). On the other hand, if you want a function which has known error properties and you may not mind oddities, the techniques suffice. Sometimes these approximations, which are much faster are useful, such as the square root function.

```
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;}
```

This is roughly four times faster than the naïve 1/sqrt(x), though it has a slightly higher relative error

Double vs Float

Can use the difference between single and double precision floating-point numbers to show errors inherent in floating-point. However, generally for practical problems, it is better to use double, almost exclusively. It has smaller errors and has often no or a very little speed penalty. However, where we have a floating-point array (where size matters and where (1) accuracy lost is manageable, (2) reduced exponent range is not a problem).

Error Accumulation

- During a computation, rounding errors accumulate, and in the worst case, they will often approach the error bounds we have calculated.
- IEEE rounding was carefully arranged to be statistically unbiased – so programs (and inputs) behave like independent random errors of mean zero.
- k-operations program produces errors of around $machineEpsilon\sqrt{k}$ rather than $machineEpsilon \times \frac{k}{2}$

Part 6: Ill-Conditionedness and Condition Number

In solving simultaneous equations, we simply want to find the intersection of the lines. Iff you have a small a, b, c, d in the matrix, as below:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} \begin{pmatrix} p \\ q \end{pmatrix} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

Then we get large values for the inverse of the matrix, therefore any small absolute errors in p or q will be greatly amplified in x and y. (This occurs when the lines are nearly parallel).

Ill-Conditioned: When a 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 value of the inputs (a, b, c, d, p, q). Such systems are called ill-conditioned. Simple example is matrices but is a problem for many real-life situations. We can get insight by calculating a bound for partial derivatives wrt the inputs, near the point in question: $\frac{\partial x}{\partial a}, \dots, \frac{\partial x}{\partial q}, \dots, \frac{\partial y}{\partial a}, \dots, \frac{\partial y}{\partial q}$

(Relative) Condition Number

The $\log_{10}$ ratio of relative error in the output to that of the input. A problem with a high condition number is said to be ill-conditioned.

Examples:

1. **A large function:**
   - **a.** F(x) = 10^{22} – well behaved
   - **b.** Cond = -inf
2. **A large derivative:**
   - **a.** F(x) = 10^{22}x
   - **b.** Cond = 0
3. **A spiking function:**
   - **a.** $F(x) = \frac{1}{0.001 + 0.999(x - 1000)}$
   - **b.** This has a very nasty pole
4. **A cancelling function:**
   - **a.** F(x) = x – 1000
   - **b.** It apparently looks well behaved but:
     - **i.** F(x(1+n)) = x(1+n) – 1000
     - **ii.** Cond = $log_{10}\left(\frac{n'}{n}\right) = \log_{10}\left(\frac{f(x(1+n)) - f(x)}{n.f(x)}\right)$
     - **iii.** $= \log_{10}\left(\frac{xn}{n(x-1000)}\right)$

L'Hôpital's Rule

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

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

Backwards Stability

Inverse algorithm exists that can accurately regenerate the input from the output. This implies Well-Conditionedness

Monte-Carlo Technique

- If formal methods are inappropriate for determining conditionedness of a problem, you can resort to probabilistic (Monte Carlo) techniques.
- **Method**
  - Take the original problem and solve
  - Then take many variants of the problem, each perturbing the value of one or more parameters or input variables by a few ulp or a fraction of a percent.
  - We then solve all of these.
  - If these all give similar solutions, then the original problem is likely to be well-conditioned.
  - If not, then there is likely to be some instability.

Adaptive Methods

- Sometimes, there are problems which are well-behaved in some regions but behaves badly in another.
- Therefore, the best way is to **discretise** the problem into small blocks which has finer discretisation in problematic areas (for accuracy) but larger in the rest (for speed)
  - Otherwise, can use a dynamically varying ΔT
- Sometimes, an iterative solution to a differential equation is fasted solved by solving for a course discretisation or large step size and then refining.
- **Simulated Annealing**
  - Standard technique – large jumps and random decisions used initially but as the temperature control parameter is reduced, the procedure becomes more conservative.

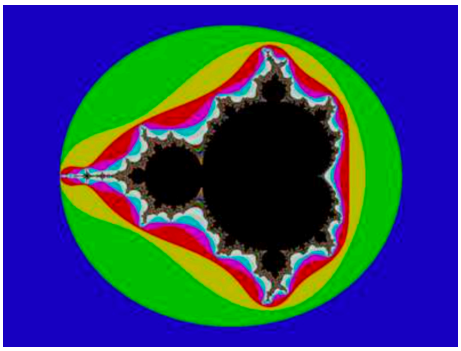
Chaotic Systems

Nastier form of ill-conditionedness for which the computed function is highly discontinuous – small input regions for which a wide range of output values occur.

Examples:

- Mandelbrot Set

  - Set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from z = 0, for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc remains bounded in absolute value.



- Verhulst's Logistic Map
  - Population changes of rabbits and foxes, with reproduction proportional to current population and starvation (growth rate will decrease at a rate proportional to a value obtained by taking the theoretical carrying capacity of the environment minus current population)

Part 7: Solving Systems of Simultaneous Equations

Gaussian Elimination

We can freely add a multiple of any row to any other, so we can do this to create an upper-triangular form and then back substitute. This is $O(n^3)$.

Minor Problems:

- We have a pivot element which we use to do the first step of Gaussian Elimination
- *First step is to multiple the top line by* $-A_{21} / A_{11}$
- A small pivot adds a lot of large numbers to the remaining rows; therefore, the original data can be lost in underflow.
- Thus, should always choose the row with the largest leading value, in order to prevent this – this is **partial row pivoting**

L/U Decomposition

In order to find a number of right-hand-sides to solve is a good approach to first find the inverse matrix?

However, in general a better approach is to first triangular decompose A = LU, then:

1. Find y from Ly = b, using forwards substitution with triangular form L
   a. L = left triangular
   b. U = upper triangular
2. Then find x from Ux = y using backwards substitution.

Dolittle Algorithm:

- We do a normal Gaussian Elimination on A to get U, but keeping track of the steps you would make on rhs in an extra matrix – this is L.
  - Write Gaussian step i on the rhs as a matrix multiply with G_i – close to identity form

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

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

Cholesky Transform

Works for symmetric +ve-definite Matrices

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

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

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

Can now solve $Ax = b$ using:

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

This has complexity $O(n^3)$ and while generally stable, the sqrt has the potential to fail. Therefore, sometimes much better to use L/D/U

Non-Hermitian Matrices

If a matrix is not symmetric, we can still use Cholesky to find B^{-1} even when B is not symmetric. We first fine the inverse of BB^T which is always symmetric. Then we can say that $B^{-1} = B^T(BB^T)^{-1}$

Hermitian Matrix: Matrix which is its own conjugate transpose

$$A = \overline{A^T}.$$

When to use what technique

1. Input is triangular?
   a. Use in that form
2. Input rows or cols can be permuted to be triangular?
   a. Use the permutation
3. Input rows and cols can be permutated to be triangular?
   a. Expensive search unless sparse
4. Array is symmetric and +ve definite?
   a. Use Cholsesky
5. None of above, but array is square
   a. One rhs – Gaussian Elimination
   b. >1 rhs – L/U Decomposition
6. Array is over-specified

   a. Regression fit
7. Array is under specified
   a. Optimise wrt some metric function

Simulated Annealing Example (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 sometimes accepted while above 100 degrees to avoid local minima
- Below 100, only accept positive progress (quenching)

Part 8: Alternative Floating-Point Technologies

If we can't find a way to compute a good approximation of the exact answer of a problem, or if we know an algorithm but don't know how the errors propagate, so the answer may be useless, then we want to do something else.

Interval Arithmetic

Represent a mathematical real number with two IEEE floating point number. One gives a representable number guaranteed to be lower or equal to the actual value and the other greater than or equal.

+:

1. It naturally copes with uncertainty in input values
2. IEEE arithmetic rounding modes (to +ve, -ve infinity) do much of the work

-:

1. It will be slower
2. Some algorithms converge in practise while the computed bounds can be far apart.
3. Need a big more work that you would expect if the range of the denominator in a division includes 0, since the output range then includes zero
4. Conditions can become both true and false!

Arbitrary Precision Arithmetic

Some packages allow you to set the precision with which you want to work with on a run-by-run basis. It is clear that you can run with any number of significant figures and still get the same errors, but they would generally arise at a different point.

Adaptive Precision: Some packages even allow adaptive precision, where you reduce the number of significant figures you work with if you find out that it is not necessary to do that.

Example: GNU MPFR library gives multiple-precision floating-point computations. Most high-level languages have a binding for access without using C or C++

Exact Real Arithmetic (CRCalc, XR, IC Reals, iRRAM)

Here, we consider using digit streams, that is lazy lists of digits, for infinite precision real arithmetic. It obviously however, has some failures – can never even get the first digit of $0.\dot{3} + 0.\dot{6}$

Part 9: FDTD (Finite-Difference, Time-Domain Simulation) and Monte Carlo Simulation

Definitions

Event-Driven: A lot of computer-based simulation uses discrete events and a time-sorted pending event queue – used for systems where activity is locally concentrated and unevenly spaced (digital logic and queues)

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

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

Ergodic: For systems where the time average is the ensemble average (do not get stuck), we can use one starting seed and average in the time domain.

Monte Carlo Example

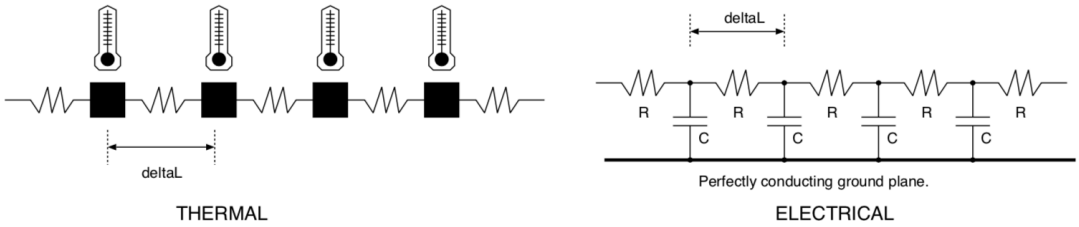
Finding Pi:

X <- random (0, 1)
Y <- random (0, 1)
If ($X^2 + Y^2 < 1$) hit++
Darts++

As darts -> ∞, $\frac{hits}{darts} \rightarrow \frac{\pi}{4}$

FDTD Simulations

1D Example (finding deltaL):



In order to do this, we split the things into finite-sized, elemental lumps:

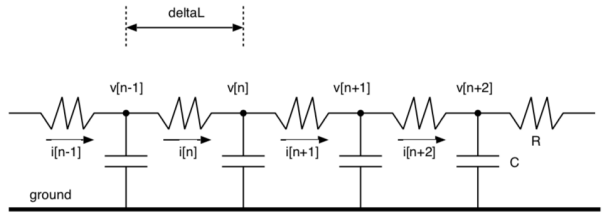
Per element	Heat Conduction in a Rod	R/C Delay Line
$\rho A = dR/dx$	thermal resistance (J/s/°C/m)	electrical resistance (Ω/m)
$\epsilon D = dC/dx$	thermal capacity (°C/J/m)	electrical capacity (F/m)
Element state	Temperature (°C)	Voltage (V)
Coupling	Heat flow (J/s)	Charge flow (Q/s=I)

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

We create a **state vector** which contains the variables whose values need to be saved from one-time stamp to the next. By changing the Δt, (and element size where relevant) we can change the accuracy.

Iterating Finite Differences

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



For each time step:

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

But, i(x) is not a state variable, so does not need to be stored, therefore (in actuality, the current will decrease as the voltages change during a time step):

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

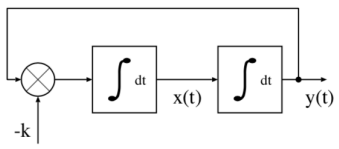
Aside: Newton's Laws of Mixing and Head Conduction

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

FDTD Example – Instrument Physical Modelling (Violin)

1. Bow: near linear velocity and static and dynamic friction coefficients
2. String: 2D or 2D simulation? Linear elemental length and string's mass per unit length and elastic modulus is required.
3. Body: A 3D resonant air chamber? Size of cubes for modelling? We also need to know the topology as well as the air's density and elastic modulus.

Example 2: SHM quadrature oscillator – generating sine and cosine waveforms



Controlling equations:

$$x = \int -ky \, dt$$
$$y = \int x \, dt$$

For each time step:

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

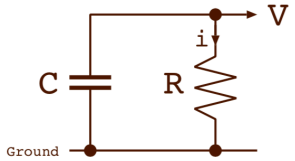
Euler Method Stability

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

- Simulation of $y = e^{-ax}$ fails for step size $h > 2a$
- Also, systems that head for a unique equilibrium point can initially use a large step size without any problems.

Forwards and Backwards Differences

- The forward difference method (Euler Method) uses the rate values at the end of one timestep as though constant in the next timestep
- We can sometimes find end rates for the current step from simultaneous equation
  - Example: Capacitor discharging



Dynamic equations:

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

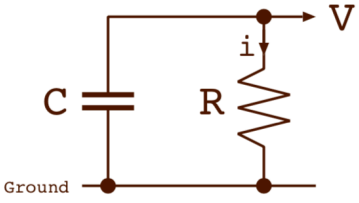
Dyns rewritten:

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

Forward iteration:

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

- This generally leads to an over-discharging error of 50ppm (second order Taylor of e^x)
- If we halve Δt, we half the error
- We can however, potentially use the ending loss rate in the 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:

$$V(n+1) := V(n) - \tau V(n+1)$$
$$= V(n)/(1+\tau)$$

Numeric result:

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

So this under-decays...

- Therefore, by combining equal amounts of the forwards and backwards stencils is a good approach – **the Crank Nicolson Method**

Stability of Interacting Elements

FDTD errors cancel out in two cases:

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

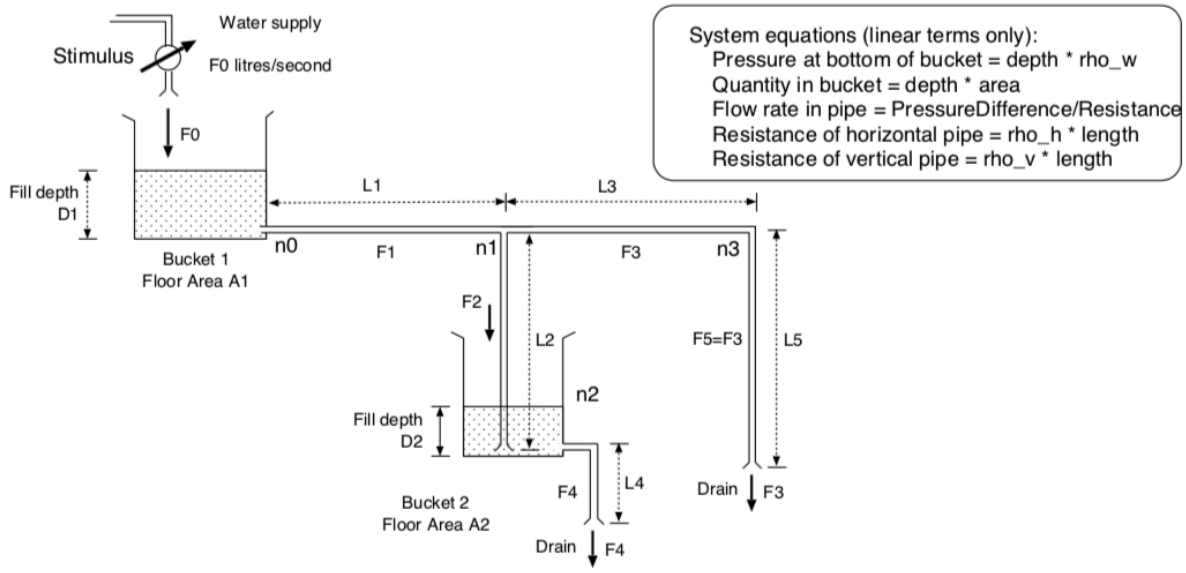
Most systems that are simulated with FDTD have interactions between element pairs in both directions that are part of a stabilising, negative feedback arrangement.

Generally, if exponential decay using forward stencil, correspondingly less decay in following time step means it will cancel out, stopping truncation error accumulation.

Part 10: Fluid Network Simulation

We can consider water flow networks to be similar to computers – they can be arranged to technically work similar to computers, with the water flowing instead of current flow through a circuit. MONIAC (Monetary National Income Analogue Computer) models the national economic processes of the UK using water.

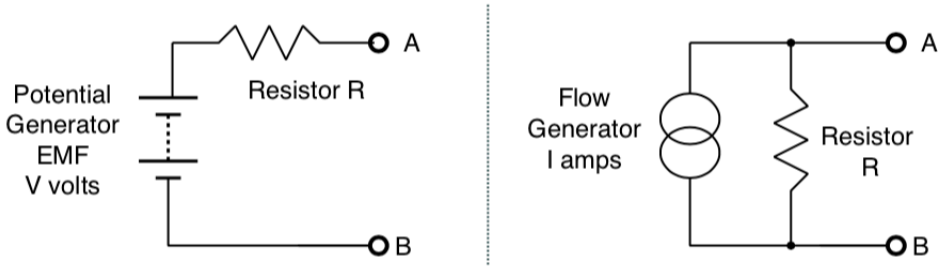
There is the problem of writing a FDTD simulation for a water circuit with non-linear components (an analytical solution of the integral equations would have to assume pipes have linear flow properties: rate is proportional to potential / pressure difference – Ohm's Law)



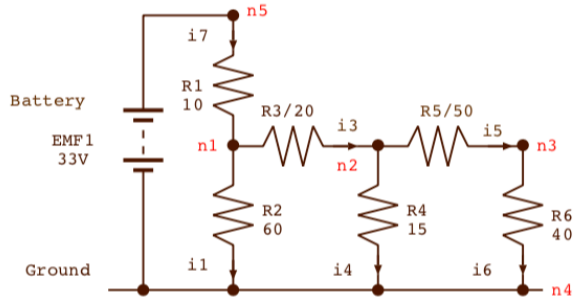
Circuit Simulation: Finding the Steady State (DC) Operating Point

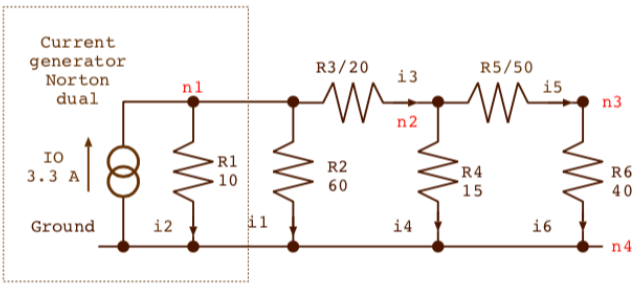
We can solve the circuit using **Nodal Analysis** – solve the set of flow equations to find node potentials. We can convert voltage generators using Norton and Thévinin Equivalents (Duals)

This states that 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.



Example:





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

Finding Steady State Operating Point

Kirchoff's Current Law gives us an equation for each node – the sum of the currents is zero.

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

This can be solved using Gaussian Elimination.

Component Templates / Patterns

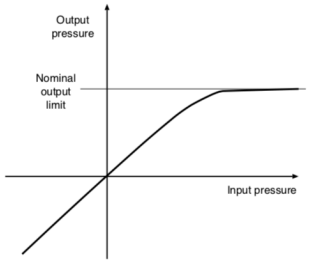
Conducting Channels: A conductance between a pair of nodes (x, y) appears four times in the conductance matrix. Twice positively on the leading diagonal at (x, x) and (y, y) and twice negatively at (x, y) and (y, x). A conductance between node x and any reference plane appears just on the leading diagonal at (x, x)

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

Constant Potential Generators: The constant potential generators must be converted into current generators using a circuit modification.

Non-Linear Components

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



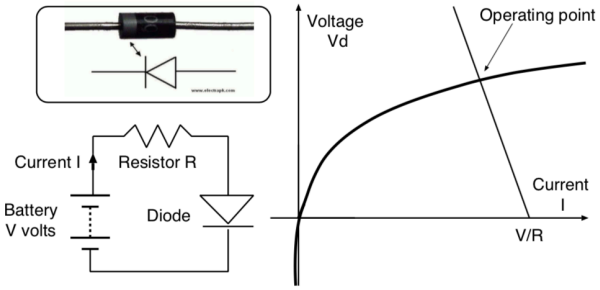
The resistance increases drastically above the target output pressure.

A Pressure Regulator Valve

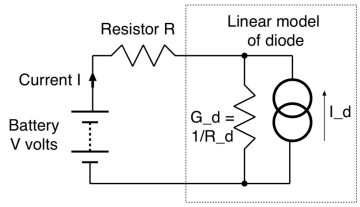
Semiconductor Diode: archetypal non-linear component. The operating point of a battery, resistor and diode circuit is given by the following equations:

$$I_d = I_S\left(e^{V_d/(nV_T)} - 1\right) \quad \text{(Shockley ideal diode equation)}$$
$$V_d = V_{\text{battery}} - I_d R \quad \text{(Composed battery and resistor equations)}$$

The diode's resistance changes according to its voltage (but a well-behaved derivative is readily available)



The way we model this is by linearizing by linearizing the component at the operating point. In the electronic example, the non-linear diode changes to a linear current generator and a conductance pair.



Gd and Id are both functions of Vd:

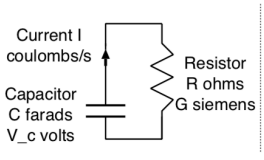
$$G_d = dI/dV = \frac{I_S}{nV_T} e^{V_d/(nV_T)}$$
$$I_d = V_d G_d$$

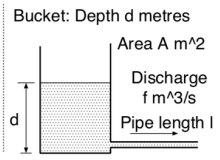
When we solve the circuit equations for a starting (Gd, Id) pair, we get a new Vd. We must iterate. Owing to the differentiable, analytic form of the semiconductor equations, Newton-Raphson iteration can be used. A typical stopping condition is when the relative voltage changes less than 10^{-4}.

Time-Varying Components

Examples: (1) Charge leaves a capacitor at a rate proportional to the leakage conductance, (2) Water flows down pipes at a rate proportional to the amount remaining in the bucket – see below

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

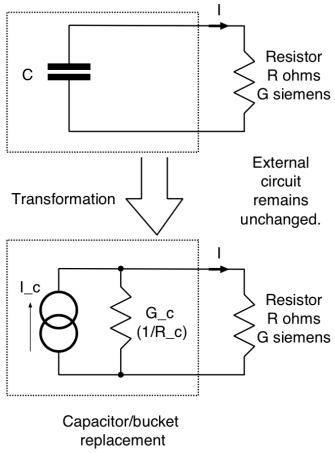




$$\begin{aligned} \mathrm{d}d/\mathrm{d}t &= -f/A \\ f &= \rho d g / k l \end{aligned}$$

In these cases, we would perform time-domain simulation with a suitable ΔT.

Then, we linearize the time-varying component using a snapshot: replace the capacitor to a resistor and current generator.



$$\begin{aligned} \frac{\mathrm{d}V}{\mathrm{d}I} &= \frac{\mathrm{d}(\Delta V)}{\mathrm{d}I} = \frac{\mathrm{d}(I\Delta T/C)}{\mathrm{d}I} = \frac{\Delta T}{C} \\ G_c &= \frac{\mathrm{d}I}{\mathrm{d}V} = \frac{C}{\Delta T} \\ I_c &= V_c G_c \end{aligned}$$

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

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

Adaptive Timestep

1. Iterate in the current timestep until convergence
2. Extrapolate forwards in the time domain using preferred stencil
    a. Could just be a simple forwards stencil

Choosing ΔT:

- Too large: errors accumulate undesirably
    - **A larger timestep will mean more iterations within a timestep since it starts with the values from the previous timestep**
    - **I don't really understand this at all**
- Too small: slow simulation

Therefore, go for an iteration count adaptive method:

- If $N_{it} > 2N_{max}$ { ΔT* = 0.5; revert_timestep(); }
- Else if $N_{it} > N_{max}$ { ΔT* = 0.9 }

- Else if $N_{it} < N_{max}$ { $\Delta T^* = 1.1$ }