3BA1 Part II — Numerical Methods

Andrew Butterfield

May 6, 2004

Contents

1 Lectures **4**
1.1 3BA1-II Lecture 1 4
1.2 3BA1-II Lecture 2 7
1.2.1 Sources of Error 7
1.2.2 Key Definitions (I) 7
1.2.3 Example 8
1.2.4 Key Definitions (II) 8
1.3 3BA1-II Lecture 3 9
1.3.1 Error Propagation 9
1.3.2 Example 10
1.3.3 (Common) Functions of two variables 10
1.3.4 The problem with subtraction 12
1.4 3BA1-II Lecture 4 13
1.4.1 Solving Quadratic Equations 13
1.4.2 Rounding Errors in Floating Point Arithmetic 16
1.5 3BA1-II Lecture 5 17
1.5.1 Arithmetic with Floating Point Numbers 17
1.6 3BA1-II Lecture 6 21
1.6.1 Accumulated Errors 21
1.6.2 IEEE Floating Point Standard 23
1.7 3BA1-II Lecture 7 24
1.7.1 Function Evaluation 24
1.7.2 Alternating Series 25
1.7.3 Bounded Monotonic Sequences 26
1.8 3BA1-II Lecture 8 28
1.8.1 An efficient implementation of $\surd$ 28
1.9 3BA1-II Lecture 9 32

1.9.1 Root Finding . . . 32
1.9.2 Generating initial guesses . . . 32
1.9.3 Interval Techniques . . . 33
1.9.4 Point Techniques . . . 34
1.10 3BA1-II Lecture 10 . . . 37
1.10.1 Analysis of Iterative Methods . . . 37
1.10.2 Convergence Analysis . . . 37
1.10.3 Detailed Analysis of Newton-Raphson . . . 40
1.10.4 Finding Roots - Summary . . . 40
1.11 3BA1-II Lecture 11 . . . 41
1.11.1 Numerical Differentiation . . . 41
1.11.2 Error Estimation . . . 41
1.11.3 Exercise . . . 42
1.12 Exercise (2004) No. 1 . . . 43
1.12.1 Objective . . . 43
1.12.2 Methodology . . . 43
1.12.3 Questions . . . 44
1.12.4 Submission . . . 44
1.13 3BA1-II Lecture 12 . . . 45
1.13.1 Error analysis of Central Difference . . . 45
1.13.2 Rounding Error in Difference Equations . . . 46
1.13.3 Richardson Extrapolation . . . 48
1.14 3BA1-II Lecture 13 . . . 50
1.14.1 Discussion of Exercise (2004) No. 1 (§1.12) . . . 50
1.15 3BA1-II Lecture 14 . . . 57
1.15.1 Solving Differential Equations Numerically . . . 57
1.15.2 The Lipschitz Condition . . . 58
1.15.3 Numerically solving $y' = f(x, y)$ . . . 59
1.16 3BA1-II Lecture 15 . . . 62
1.16.1 Improved Differentiation Techniques . . . 62
1.16.2 Interpolation . . . 63
1.17 3BA1-II Lecture 16 . . . 67
1.17.1 Numerical Integration . . . 67
1.17.2 Trapezoidal Rule ($k = 1$) . . . 68
1.17.3 Simpson's Rule ($k = 2$) . . . 69
1.17.4 Romberg's Method . . . 70

A Calculus Prerequisites **71**
A.1 Limits and continuity . . . 71
A.1.1 Intermediate Value Theorem . . . 72
A.1.2 Extreme Value Theorem . . . 72
A.2 Differentiable Functions . . . 72
A.2.1 Rolle's Theorem . . . 73
A.2.2 Mean Value Theorem . . . 73
A.2.3 Generalised Rolle's Theorem . . . 73
A.3 Integrals . . . 73
A.3.1 First Fundamental Theorem . . . 73
A.3.2 Second Fundamental Theorem . . . 74
A.3.3 Mean Value Theorem for Integrals . . . 74
A.3.4 Weighted Interval Mean Value Theorem . . . 74
A.4 Series . . . 74
A.4.1 Taylor's Theorem . . . 74
A.5 Evaluation of a Polynomial . . . 75
A.5.1 Horner's Method for Polynomial Evaluation . . . 75
A.6 Taylor's Theorem revisited . . . 75
A.7 Laws of Differentiation . . . 76

B Haskell Experimentation for 3BA1 Part II **77**
B.1 Representation Errors . . . 77
B.1.1 Associativity . . . 77
B.1.2 Repeated Summing . . . 78
B.1.3 Root Finding . . . 78
B.1.4 Difference Equations . . . 82
B.1.5 Full Richardson Extrapolation . . . 84
B.1.6 Solving the Diffusion Equation . . . 85

Chapter 1

Lectures

1.1 3BA1-II Lecture 1

This course concerns itself with performing arithmetic with real numbers, using computing machinery. The key issue is that computer arithmetic necessarily involves finite approximations to infinite quantities. Scientific and Engineering theories are built on real numbers — the differential and integral calculi used so extensively all depend on real numbers for their soundness.

Consider the various number types:

$$\begin{array}{rcc} \textit{Naturals} & \mathbb{N} & 0, 1, 2, 3, \ldots \\ \textit{Integers} & \mathbb{Z} & \ldots, -2, -1, 0, 1, 2, \ldots \\ \textit{Rationals} & \mathbb{Q} & 0, \pm 1/1, \pm 1/2, \pm 2/1, \pm 1/3, \pm 3/1, \pm 2/3, \pm 3/2, \ldots \end{array}$$

Each of these contains an infinite number of numbers, but each number is itself *finite*. However, the reals are different:

$$\textit{Reals} \quad \mathbb{R}$$

Each real number is itself an infinite object:

$$\begin{array}{rcl} 1 & = & 1.0000000000000000\ldots \\ 1/3 & = & 0.33333333333333333333\ldots \\ \pi & = & 3.15159\ldots \\ e & = & 2.717\ldots \end{array}$$

We can only handle finite quantities in reality, either when using computers, or calculating by hand !

Consider the simple law of the associativity of addition:

$$a + (b + c) = (a + b) + c$$

This holds for all numbers, whether natural, integer, rational, or real. But what happens if we are representing real numbers using some finite encoding scheme such as floating point ?

Consider a scheme which represents real numbers using 3 decimal digits, plus an decimal exponent in the range $-9, \ldots, +9$:

$$\pm d_0.d_1 d_2 d_3 \cdot 10^e$$

where $d_i \in 0, 1, \ldots, 9$, $d_0 \neq 0$ and $-9 \leq e \leq 9$. Let $a = 1.000 \cdot 10^0$ and $b = c = 3.000 \cdot 10^{-4}$. Consider $a + b$:

$$\begin{array}{ll}
 & a + b \\
= & \quad \text{“ values of } a \text{ and } b \text{ ”} \\
 & 1.000 \cdot 10^0 + 3.000 \cdot 10^{-4} \\
= & \quad \text{“ align and extend digits, noting largest exponent ”} \\
 & 1.000\,0 + 0.000\,3 \qquad 10^0 \\
= & \quad \text{“ add ”} \\
 & 1.000\,3 \qquad 10^0 \\
= & \quad \text{“ round to 3 decimals, and link in exponent ”} \\
 & 1.000 \cdot 10^0
\end{array}$$

So we have $a + b = a$ even though $b \neq 0$!. We can also see clearly that $(a + b) + c$ will also equal a in this case.

Now consider $b + c$:

$$\begin{array}{ll}
 & b + c \\
= & \quad \text{“ values of } b \text{ and } c \text{ ”} \\
 & 3.000 \cdot 10^{-4} + 3.000 \cdot 10^{-4} \\
= & \quad \text{“ align and extend digits, noting largest exponent ”} \\
 & 3.000\,0 + 3.000\,0 \qquad 10^{-4} \\
= & \quad \text{“ add ”} \\
 & 6.000\,0 \qquad 10^{-4} \\
= & \quad \text{“ round to 3 decimals, and link in exponent ”} \\
 & 6.000 \cdot 10^{-4}
\end{array}$$

We now add a and $b+c$:

$$\begin{array}{ll} & a+(b+c) \\ = & \quad \text{“ values of } a \text{ and } b+c \text{ ”} \\ & 1.000 \cdot 10^0 + 6.000 \cdot 10^{-4} \\ = & \quad \text{“ align and extend digits, noting largest exponent ”} \\ & 1.000\,0 + 0.000\,6 \qquad 10^0 \\ = & \quad \text{“ add ”} \\ & 1.000\,6 \qquad 10^0 \\ = & \quad \text{“ round to 3 decimals, and link in exponent ”} \\ & 1.001 \cdot 10^0 \end{array}$$

So for this floating point arithmetic scheme we get

$$\begin{array}{c} 1.000 \cdot 10^0 + (3.000 \cdot 10^{-4} + 3.000 \cdot 10^{-4}) \\ = \\ 1.001 \cdot 10^0 \\ \neq \\ 1.000 \cdot 10^0 \\ = \\ (1.000 \cdot 10^0 + 3.000 \cdot 10^{-4}) + 3.000 \cdot 10^{-4} \end{array}$$

We have seen that in general for finite floating point arithmetic that in general

$$\begin{array}{rcl} a+(b+c) & \neq & (a+b)+c \\ b \neq 0 & \not\Rightarrow & a+b \neq a \end{array}$$

From our example above we can see that the best approach when adding numbers together is to add the smallest first, were this information is available.

Consider the following sum:

$$\sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \cdots \frac{1}{n}$$

The best way to compute this is to start with $k = n$ and work down to 1, adding the smallest numbers first:

```
s :=0 ;
for( k=n, k>=1; k--)
  s += 1/k ;
```

Numerical Analysis and Methods are the “science” of performing these numerical calculations in a fashion that ensures that accuracy is maintained with algorithms that are also efficient.

1.2 3BA1-II Lecture 2

1.2.1 Sources of Error

- Input data error — measurement error, or reals reported with fixed no. of digits
- rounding error — computation using fixed number of digits
- truncation error — replacing infinite process by a finite one (infinite series by partial sum, or function by polynomial approximation.

1.2.2 Key Definitions (I)

Exact value a — e.g. $a = \sqrt{2}$.

Approximation $\bar{a}$ — e.g. $\bar{a} = 1.414$.

Absolute error in $\bar{a}$ $\Delta a = \bar{a} - a$, or $\bar{a} = a + \Delta a$ — e.g. $\Delta a = -0.0002135\ldots$.

Absolute error (Magnitude) $|\Delta a|$ — e.g. $|\Delta a| \leq 0.00022 \leq 0.0003$ (always round up).

Relative error in $\bar{a}$ $\frac{\Delta a}{a}$, $(a \neq 0)$ — e.g. $\frac{\Delta a}{a} = \frac{-0.0002135}{\sqrt{2}} \approx -0.000151011\ldots$

Relative error (Magnitude) $\left|\frac{\Delta a}{a}\right|$, $(a \neq 0)$ — e.g. $\left|\frac{\Delta a}{a}\right| \leq 0.00016 \leq 0.0002$ (again, always round up).

The following 3 statements are equivalent:

$$\begin{aligned} \bar{a} &= 1.414, \quad |\Delta a| \leq 0.22 \cdot 10^{-3} \\ a &= 1.414 \pm 0.22 \cdot 10^{-3} \\ 1.41378 \leq\ & a \leq 1.41422 \end{aligned}$$

Decimal Digits are the digits to the right of the decimal point.

Chopping to t decimal digits means dropping all digits after the tth (error: $\leq 10^{-t}$).

Rounding to even to t (decimal) digits means that if the number to the right of the tth digit is less that $0.5 \cdot 10^{-t}$, we chop, if greater, we increase the tth digit by 1, and if equal, we chop if the tth digit is even, and increase by 1 if odd (error: $\leq 0.5 \cdot 10^{-t}$).

If approximate value is rounded or chopped, then add in that error.

1.2.3 Example

Consider the following example:

$$\begin{aligned} & b & = & \ 11.2376 \pm 0.1 \\ 11.1376 \ \leq & \ b & \leq & \ 11.3376 \end{aligned}$$

As the error is in the first decimal digit it makes no sense to keep the 3 least significant digits in the approximation, so we round to one decimal digit:

$$b_{rounded} = 11.2$$

It is *not* the case that $11.1 \leq b \leq 11.3$ — we cannot use original error bound alone with rounded number. We need to estimate the rounding error $|R_B|$:

$$\begin{aligned} |R_B| &= \left|b_{rounded} - \bar{b}\right| \\ &= |11.2 - 11.2376| \\ &= 0.0376 \\ &< 0.04 \end{aligned}$$

Again there is little point having a rounding error with so many significant digits, so we round it *up* to one significant digit. We round up (and not "to even") to be safe — overestimating errors is safer than underestimating them.

We obtain our sensible approximation of b with error bounds by adding the original error bound (± 0.1) magnitude to that rounding error (± 0.04) magnitude:

$$\begin{aligned} & b & = & \ 11.2 \pm 0.14 \\ 11.06 \ \leq & \ b & \leq & \ 11.34 \end{aligned}$$

1.2.4 Key Definitions (II)

A number has t **correct decimals**, if $|\Delta a| \leq 0.5 \cdot 10^{-t}$. Note a number only has correct decimals if the error is less than 0.5.

In a number where $|\Delta a| \leq 0.5 \cdot 10^{e}$, then a **significant digit** is one which is not a leading zero and whose unit is greater than or equal to 10^{e}.

Examples

Approx	Correct Decimals	Significant Digits
$0.00065437 \pm 0.5 \cdot 10^{-6}$	6	3
312.538 ± 0.5^{-2}	2	5
675000 ± 500		3

1.3 3BA1-II Lecture 3

1.3.1 Error Propagation

Consider feeding a number $x = \bar{x} \pm \epsilon$ into a function f. What is the error bound on the result ? (We assume for now that we can calculate f with perfect accuracy).

We can define the function error at $\bar{x}$ as

$$\Delta f = f(\bar{x}) - f(x)$$

However, usually we only know bounds on the error ($\pm\epsilon$). How do we get an error estimate?

A simple answer can be given if the function is monotonic over an interval including the approximation and all error values. Given knowledge of $\bar{x}$ and ϵ, then the range of values that x could have is

$$\bar{x} - \epsilon \leq x \leq \bar{x} + \epsilon$$

If the function is monotone over this interval, then the largest and smallest values that f takes must be at each end of the interval, so we can calculate:

$$\begin{aligned} |\Delta f| &= |f(\bar{x}) - f(x)| \\ &\leq \max\left\{ \begin{array}{c} |f(\bar{x}+\epsilon) - f(\bar{x})| \\ |f(\bar{x}-\epsilon) - f(\bar{x})| \end{array} \right\} \end{aligned}$$

How do we deal with more general functions ? If they are non-monotonic in general, but are differentiable, then we can use the **Mean Value Theorem**:

If f is continuous over interval (a, b), then there exists c, with $a \leq c \leq b$ such that

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

We apply the theorem with $a = x$, $b = \bar{x}$ and replacing c by ξ.

$$\begin{aligned} f'(\xi) &= \frac{f(\bar{x}) - f(x)}{\bar{x} - x} \\ &= \frac{\Delta f}{\Delta x} \\ &\Downarrow \\ \Delta f &= f'(\xi)\Delta x \end{aligned}$$

for some ξ between x and $\bar{x}$.

In calculations, we don't know ξ, so we use $\bar{x}$ as an approximation, and compute magnitudes:

$$\bar{\Delta f} \leq f'(\bar{x})\,|\Delta x|$$

If our number is expressed as $\bar{x} \pm \epsilon$, then this formula becomes:

$$|\Delta f| \leq f'(\bar{x})\epsilon$$

We usually then add something to the error bound for safety.

1.3.2 Example

$$\begin{aligned}
f(x) &= \sqrt(x) \\
a &= 2.05 \pm 0.01 \\
\Delta f &= f'(\xi)\Delta a \\
&= \frac{1}{2\sqrt{\xi}}\Delta a \\
|\Delta f| &\lesssim \frac{1}{2 \cdot \sqrt{2.05}}\,|\Delta a| \\
&\leq \frac{0.01}{2 \cdot \sqrt{2.05}} \\
&\leq 0.0036 \\
&\leq 0.04
\end{aligned}$$

If we compute $\sqrt{2.05}$ we get $1.4317821063276353154439601231034\ldots$ so we can express our result as

$$\sqrt{2.05 \pm 0.01} = 1.43 \pm 0.04$$

1.3.3 (Common) Functions of two variables

We now consider the error propagation properties of addition, subtraction, multiplication and division.

Let $y = x_1 + x_2$, and assume we know $\bar{x_1}$ and $\bar{x_2}$. Then

$$\Delta y = \bar{y} - y = \bar{x_1} + \bar{x_2} - x_1 - x_2 = \Delta x_1 + \Delta x_2$$

If we only know bounds for absolute error, then we obtain:

$$|\Delta y| = |\Delta x_1 + \Delta x_2| \leq |\Delta x_1| + |\Delta x_2|$$

For $y = x_1 - x_2$ we get

$$\Delta y = \Delta x_1 - \Delta x_2$$

and

$$|\Delta y| \leq |\Delta x_1| + |\Delta x_2|$$

In general, if $y = \sum_{i=1}^{n} x_i$ we get

$$|\Delta y| \leq \sum_{i=1}^{n} |\Delta x_i|$$

For multiplication ($y = x_1 x_2$):

$$\begin{aligned}\Delta y &= \bar{x_1}\bar{x_2} - x_1 x_2 = (x_1 + \Delta x_1)(x_2 + \Delta x_2) - x_1 x_2 \\ &= x_1 \Delta x_2 + x_2 \Delta x_1 + \Delta x_1 \Delta x_2\end{aligned}$$

We can disregard the last term if errors are small, and get relative errors:

$$\frac{\Delta y}{y} \approx \frac{\Delta x_1}{x_1} + \frac{\Delta x_2}{x_2}$$

and taking absolute values:

$$\left|\frac{\Delta y}{y}\right| \lesssim \left|\frac{\Delta x_1}{x_1}\right| + \left|\frac{\Delta x_2}{x_2}\right|$$

For division:

$$\begin{aligned}y &= x_1/x_2 \\ \Delta y &= \bar{x_1}/\bar{x_2} - x_1/x_2 \\ &= (x_1 + \Delta x_1)/(x_2 + \Delta x_2) - x_1/x_2 \\ &= \frac{(x_1 + \Delta x_1)x_2 - x_1(x_2 + \Delta x_2)}{(x_2 + \Delta x_2)x_2} \\ &= \frac{x_1 x_2 + x_2 \Delta x_1 - x_1 x_2 - x_1 \Delta x_2}{x_2 x_2 + \Delta x_2 x_2} \\ &= \frac{x_2 \Delta x_1 - x_1 \Delta x_2}{x_2 x_2 + \Delta x_2 x_2} \\ &\approx \frac{x_2 \Delta x_1 - x_1 \Delta x_2}{x_2 x_2} \\ \frac{\Delta y}{y} &\approx \frac{x_2 \Delta x_1 - x_1 \Delta x_2}{x_2 x_2 (x_1/x_2)} \\ &= \frac{x_2 \Delta x_1 - x_1 \Delta x_2}{x_2 x_1} \\ &= \frac{x_2 \Delta x_1}{x_1 x_2} - \frac{x_1 \Delta x_2}{x_1 x_2} \\ \frac{\Delta y}{y} &\approx \frac{\Delta x_1}{x_1} - \frac{\Delta x_2}{x_2} \\ \left|\frac{\Delta y}{y}\right| &\lesssim \left|\frac{\Delta x_1}{x_1}\right| + \left|\frac{\Delta x_2}{x_2}\right|\end{aligned}$$

In general

So, for multiplicative operators we wind that relative error magnitudes add, whereas for additive operators it is absolute error magnitudes that add.

When dealing with mixed expressions (like polynomials), and in general situations, which errors are more important: absolute or relative ?

The answer depends on the specific application, but in most cases, we find that relative errors are more useful that absolute ones. We will also find that floating point systems act to maintain predictable levels of relative error magnitude.

1.3.4 The problem with subtraction

Subtracting is a problem (either subtracting two quantities with the same sign, or adding two quantities with different signs) if the magnitudes of the quantities are close:

$$\begin{aligned}
x_1 &= 10.123455 \pm 0.5 \cdot 10^{-6} \\
x_2 &= 10.123789 \pm 0.5 \cdot 10^{-6} \\
\left|\frac{\Delta x_i}{x_i}\right| &\leq \pm 0.5 \cdot 10^{-7} \\
y &= x_1 - x_2 \\
&= -0.000334 \pm 10^{-6} \\
|\Delta y|\, y &\leq \frac{10^{-6}}{0.000334} < 3 \cdot 10^{-3}
\end{aligned}$$

We see that subtracting two high-precision numbers with similar magnitudes results in a value whose (relative) precision is much smaller.

By “subtraction” is meant any additive operation were the signs of the values are such that the underlying numbers get subtracted — i.e. when subtracting two numbers of the same sign, or adding two numbers of different sign.

1.4 3BA1-II Lecture 4

1.4.1 Solving Quadratic Equations

Consider

$$ax^2 + bx + c = 0$$

with solutions

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

Try the case $a = 1, b = 18, c = 1, \quad i.e. x^2 + 18x + 1 = 0$. The first solution is

$$x_1 = \frac{-18 + \sqrt{18^2 - 4}}{2} = -9 + \sqrt{80}$$

The second solution is

$$x_2 = \frac{-18 - \sqrt{18^2 - 4}}{2} = -9 - \sqrt{80}$$

where $\sqrt{80} = 8.9943 \pm 0.5 \cdot 10^{-4}$. When we calculate values we get

$$\begin{aligned} x_1 &= -0.0057 \pm 0.5 \cdot 10^{-4} \\ x_2 &= -17.9943 \pm 0.5 \cdot 10^{-4} \end{aligned}$$

While the absolute error in each case is the same $(0.5 \pm 0.5 \cdot 10^{-4})$, the relative error for x_1 is much larger than for x_2, with x_1 having only 2 significant digits. When $|b| \approx \sqrt{b^2 - 4ac}$, we find that one solution is OK, as addition occurs, but the other involves subtraction. We can get an alternative solution which allows us to avoid such subtractions:

$$\begin{aligned}
& \frac{-b - \sqrt{b^2 - 4ac}}{2a} \\
= \quad & \text{“ multiply above and below by other solution ”} \\
& \frac{-b - \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{2a}{-b + \sqrt{b^2 - 4ac}} \\
= \quad & \text{“ cancel } 2a\text{, and multiply out top line ”} \\
& \frac{b^2 - b\sqrt{b^2 - 4ac} + b\sqrt{b^2 - 4ac} - (b^2 - 4ac)}{2a(-b + \sqrt{b^2 - 4ac})} \\
= \quad & \text{“ simplify topline ”} \\
& \frac{4ac}{2a(-b + \sqrt{b^2 - 4ac})} \\
= \quad & \text{“ Cancel } -2a \text{ ”} \\
& \frac{-2c}{b - \sqrt{b^2 - 4ac}}
\end{aligned}$$

So an alternative way to compute the solutions is:

$$x = \frac{-2c}{b \mp \sqrt{b^2 - 4ac}}$$

We can now use the standard equation for x_2 and the alternative version for x_1:

$$x_1 = \frac{-2}{18 + \sqrt{18^2 - 4}} = \frac{-1}{9 + \sqrt{80}} = \frac{-1}{17.9943 \pm 0.5 \cdot 10^{-4}}$$

We can then get the reciprocal $(0.0555731537\ldots)$ and give the result as

$$x_1 = 5.5573 \cdot 10^{-2} \pm 0.5 \cdot 10^{-6}$$

The relative error after division is the same as that before, given that -1 is known here with perfect accuracy.

Summary So, to solve $ax^2 + bx + c = 0$, we have two possibilities:

$$\begin{array}{lllllll} b \le 0: & x_1 & = & \dfrac{-b + \sqrt{b^2 - 4ac}}{2a} & x_2 & = & \dfrac{-2c}{b - \sqrt{b^2 - 4ac}} \\ \\ b \ge 0: & x_1 & = & \dfrac{-2c}{b + \sqrt{b^2 - 4ac}} & x_2 & = & \dfrac{-b - \sqrt{b^2 - 4ac}}{2a} \end{array}$$

Representation of Numbers in Computers

Decimal number system is a **position system** with **base 10**.
Let β be natural number greater than 1

$$\begin{array}{rcl} \beta & : & \mathbb{N} \\ \beta & \ge & 2 \end{array}$$

Any real number can be written:

$$(\ \pm d_n d_{n-1} \ldots d_1 d_0 . d_{-1} d_{-2} \ldots\ )_\beta$$

where each d_i is a digit between 0 and $\beta - 1$:

$$\begin{array}{rcccl} & & d_i & : & \mathbb{N} \\ 0 & \le & d_i & < & \beta \end{array}$$

The value of such a number is

$$\pm \left(d_n \cdot \beta^n + d_{n-1} \cdot \beta^{n-1} + \ldots + d_1 \cdot \beta^1 + d_0 \cdot \beta^0 + d_{-1} \cdot \beta^{-1} + d_{-2} \cdot \beta^{-2} + \ldots\right)$$

Real processor used fixed world length (typically 32, 64 bits).
Fixed point representation:

$$(\ \pm d_n d_{n-1} \ldots d_1 d_0 . d_{-1} d_{-2} \ldots\ d_m)_\beta$$

Difficult to handle both large and small numbers with this method. Consider a decimal system with $n = 2, m = 3$. The numbers we get range from 000.001 to 999.999, all viewed as accurate to $\pm 0.5 \cdot 10^{-3}$. However the corresponding relative error ranges from about $\pm 0.5 \cdot 10^{-6}$ (for 999.999) up to $\pm 0.5 \cdot 10^{-0}$ (for 000.001).

Small numbers lack significant digits, so relative error gets larger as number get smaller.

A floating point representation aims to get around this relative error problem, by enabling us to represent small quantities with as many signifcant digits as large ones have.

Floating point representation (infinite):

$$\begin{array}{rcl} X & = & M\beta^e \\ M & = & \pm D_0.D_1D_2D_3\dots \\ 0 \ leq & D_i & < \ \beta \\ D_0 & \neq & 0 \end{array}$$

Floating point representation (finite):

$$\begin{array}{rcl} x & = & m\beta^e \\ m & = & \pm d_0.d_1d_2\dots d_t \\ 0 \ leq & d_i & < \ \beta \\ d_0 & \neq & 0 \end{array}$$

Here m is M rounded to $t+1$ digits. We **normalise**, so $1 \leq |m| < \beta$. We give limits to e:

$$L \leq e \leq U$$

If result requires $e > U$, we have **overflow**.

If result requires $e < L$, we have **underflow**.

The **floating point system** (β, t, L, U) is the set of normalised floating point numbers, satisfying:

$$\begin{array}{rcccl} & x & = & m\beta^e & \\ & m & = & \pm d_0.d_1d_2\dots d_t & \\ 0 & leq & d_i & < & \beta \\ & d_0 & \neq & 0 & \\ 1 & \leq & |m| & < & \beta \\ L & \leq & e & \leq & U \end{array}$$

Examples

	β	t	L	U
IBM 3090	16	5	-65	$+62$
IEEE Single Precision	2	23	-126	$+127$
IEEE Single Precision Extended	2	≥ 32	≤ -1023	≥ 1024
IEEE Double Precision	2	52	-1022	$+1023$
IEEE Double Precision Extended	2	≥ 63	≤ -1023	≥ 1024

1.4.2 Rounding Errors in Floating Point Arithmetic

Assume x can be written exactly as $m\beta^e$.

Let $x_r = m_r\beta^e$, where m_r is m rounded to $t+1$ digits.

$$\begin{aligned}
|m_r - m| &\leq \frac{1}{2}\beta^{-t} \\
|x_r - x| &\leq \frac{1}{2}\beta^{-t}\beta^e \\
\frac{|x_r - x|}{|x|} &\leq \frac{\frac{1}{2}\beta^{-t}\beta^e}{|m|\,\beta^e} \\
&= \frac{\frac{1}{2}\beta^{-t}}{|m|} \\
&\leq \quad \text{“ normalisation: } 1 \leq |m| \text{ ”} \\
&\quad \frac{1}{2}\beta^{-t}
\end{aligned}$$

The **relative rounding error** is estimated as

$$\frac{|x_r - x|}{|x|} \leq \mu$$

where $\mu = \frac{1}{2}\beta^{-t}$ is the **unit roundoff**.

All (normalised) numbers in floating point representations have the same relative accuracy.

1.5 3BA1-II Lecture 5

1.5.1 Arithmetic with Floating Point Numbers

We want to ensure, when implementing floating point arithmetic, that the relative error in the result is less than the unit roundoff error (μ):

$$\mu = \frac{1}{2}\beta^{-t}$$

We shall take the following floating point system, with 3 decimal digits, as a working example:

$$\boxed{(\beta, t, L, U) = (10, 3, -9, +9)}$$

The value (x) of the number with mantissa m and exponent e is

$$x = m \cdot \beta^e$$

Remember that we have the following conditions:

$$\begin{array}{ccccc} L & \leq & e & \leq & U \\ 1 & \leq & |m| & < & \beta \end{array}$$

The mantissa has length $t+1$, and the first digit (the one before the decimal point) is always 1 or greater.

The key idea: internally we use longer mantissas in order to retain accuracy. We shall convert input numbers to the longer format, do the calculation in that format, and then round the result to the original length before outputting it.

$$\begin{tikzcd}[row sep=small] \text{input f.p. numbers } (\beta, t, L, U) \arrow[d] \\ \text{expand } t+1 \text{ digits to ?? digits ;} \arrow[d] \\ \text{compute} \arrow[d] \\ \text{round result to } t+1 \text{ digits} \arrow[d] \\ \text{output f.p. number } (\beta, t, L, U) \end{tikzcd}$$

To how many digits must we expand in order to not lose any accuracy ?

It can be shown that using $2t+4$ digits will allow us to evaluate the standard four operations ($+, -, \times, \div$) to unit roundoff accuracy. Consider that multiplying two $t+1$ digit strings will give an answer that requires up to $2t+2$ digits. The extra two digits are required to keep errors smaller than the rounding error.

We now consider each operation in turn.

In each case, we assume we are calculating $z = x \odot y$ where

$$\begin{array}{rcl} x & = & m_x \cdot \beta^{e_x} \\ y & = & m_y \cdot \beta^{e_y} \\ z & = & m_z \cdot \beta^{e_z} \end{array}$$

The goal is, given m_x, e_x, m_y, e_y to determine m_z, e_z. We assume that m_x and m_y have been expanded to $2t + 4$ digits by adding $t + 3$ zeros on the right.

Floating Point Addition/Subtraction

As in computer integer arithmetic, we handle addition and subtraction together. We assume $e_x \geq e_y$ to simplify the presentation (if not, swap them around):

```
$Add/Sub(m_x, e_x, m_y, e_y)$
$\widehat{=}$
  $e_z := e_x$;
  **if** $e_x - e_y \geq t + 3$
  **then**
    $m_z := m_x$
  **else**
    $m_y :=$ shift $m_y$ $(e_x - e_y)$ digits to the right;
    $m_z := m_x \pm m_y$
  **endif**;
  $TidyUp(m_z, e_z)$
```

We need to tidy-up the result by rounding to $t+1$ digits, and ensuring the result in normalised $(1 \leq |m_z| < \beta)$.

Floating Point Multiplication

Multiplication is very straightforward:

```
$Mul(m_x, e_x, m_y, e_y)$
$\widehat{=}$
  $e_z := e_x + e_y$;
  $m_z := m_x \times m_y$;
  $TidyUp(m_z, e_z)$
```

Floating Point Division

So is division:

```
$Div(m_x, e_x, m_y, e_y)$
$\widehat{=}$
  $e_z := e_x - e_y$;
  $m_z := m_x \div m_y$;
  $TidyUp(m_z, e_z)$
```

Tidying Up

We have to take the mantissa from the calculation of length $2t+4$ and round it down to the output length of $t+1$, and also ensure that the number is normalised. We normalise first, before rounding (why ?).

```
$TidyUp(m, e)$
$\widehat{=}$
  **if** $|m| > \beta$
  **then**
    Shift $m$ right one place;
    $e := e + 1$
  **else**
    **while** $|m| < 1$
    Shift $m$ left one place;
    $e := e - 1$
    **endwhile**
  **endif**;
  Round $m$ to $t + 1$ digits;
  **if** $|m| = \beta$
  **then**
    Shift $m$ right one place;
    $e := e + 1$
  **endif**;
  $Finalise(m, e)$
```

Once we have normalised our number, we need to ensure that it is actually representable in our floating point scheme:

```
$Finalise(m, e)$
$\widehat{=}$
  **if** $e > U$
  **then**
    $OVERFLOW$
  **elsif** $e < L$
  **then**
    $UNDERFLOW$;
    $z := 0$
  **else**
    $z := m \cdot \beta^e$
  **endif**
```

Here we have set $z = 0$ in the case of underflow. Another option is to return the number in un-normalised form. Then we find that the relative error is now greater that the unit roundoff, but less than it would be if we simply returned zero (in which case the relative error is unity).

For example, if the answer obtained was

$$1.234 \cdot 10^{-11}$$

we could return this un-normalised as

$$0.012 \cdot 10^{-9}$$

However we can see that our result now has only 2 significant digits, rather than 4.

1.6 3BA1-II Lecture 6

Our key result is that for any operation $\odot$, where $\odot = +, -, \times, \div$, that the floating point version can be implemented with error bounded by the unit roundoff:

$$\left| \frac{x \odot y - \mathrm{fl}[x \odot y]}{x \odot y} \right| \leq \mu = \frac{1}{2}\beta^{-t}$$

An alternative formulation expresses this in terms of ϵ, where $|\epsilon| \leq \mu$:

$$\boxed{\mathrm{fl}[x \odot y] = (x \odot y)(1 + \epsilon)}$$

This makes it easier to perform certain forms of analysis.

1.6.1 Accumulated Errors

Consider summing some series:

$$S = \sum_{k=1}^{n} x + k$$

using floating point addition.
Let S_i denote the (partial sum) result of adding the first i terms

$$S_i = \sum_{k=1}^{i} x + k$$

and we use $\hat{S}_i$ to denote the result of computing S_i using floating point arithmetic with its errors.

Consider the first few sums:

$$\begin{aligned}
\hat{S}_1 &= x_1 \\
\hat{S}_2 &= (\hat{S}_1 + x_2)(1+\epsilon_2) \\
&= (x_1 + x_2)(1+\epsilon_2) \\
\hat{S}_3 &= (\hat{S}_2 + x_3)(1+\epsilon_3) \\
&= ((x_1 + x_2)(1+\epsilon_2) + x_3)(1+\epsilon_3) \\
&= x_1(1+\epsilon_2)(1+\epsilon_3) + x_2(1+\epsilon_2)(1+\epsilon_3) + x_3(1+\epsilon_3) \\
\hat{S}_4 &= (\hat{S}_3 + x_4)(1+\epsilon_4) \\
&= ((x_1(1+\epsilon_2)(1+\epsilon_3) + x_2(1+\epsilon_2)(1+\epsilon_3) + x_3(1+\epsilon_3)) + x_4)(1+\epsilon_4) \\
&= x_1(1+\epsilon_2)(1+\epsilon_3)(1+\epsilon_4) \\
&+ x_2(1+\epsilon_2)(1+\epsilon_3)(1+\epsilon_4) \\
&+ x_3(1+\epsilon_3)(1+\epsilon_4) \\
&+ x_4(1+\epsilon_4) \\
\vdots \\
\hat{S}_n &= (\hat{S}_{n-1} + x_n)(1+\epsilon_n) \\
&= x_1(1+\epsilon_2)(1+\epsilon_3)\cdots(1+\epsilon_n) \\
&+ x_2(1+\epsilon_2)(1+\epsilon_3)\cdots(1+\epsilon_n) \\
&+ x_3(1+\epsilon_3)\cdots(1+\epsilon_n) \\
\vdots \\
&+ x_i(1+\epsilon_i)\cdots(1+\epsilon_n) \\
\vdots \\
&+ x_n(1+\epsilon_n)
\end{aligned}$$

We see that terms summed earlier have larger relative errors, so if $i < j$ then we have

$$\begin{aligned}
\hat{x}_i &= x_i(1+\epsilon_i)\cdots(1+\epsilon_{j-1})(1+\epsilon_j)(1+\epsilon_{j+1})\cdots(1+\epsilon_n) \\
\hat{x}_j &= x_i(1+\epsilon_j)(1+\epsilon_{j+1})\cdots(1+\epsilon_n)
\end{aligned}$$

This is why it is better to add series starting with the smallest terms first. These suffer the greatest relative error, but since they are small it contributes much less to the overall absolute error.

$$\begin{aligned}
\hat{x}_i &= x_i(1+\epsilon_i)(1+\epsilon_{i+1})\cdots(1+\epsilon_{n-1})(1+\epsilon_n) \\
\Delta x_i &= \hat{x}_i - x_i \\
&= \underbrace{x_i\underbrace{((1+\epsilon_i)(1+\epsilon_{i+1})\cdots(1+\epsilon_{n-1})(1+\epsilon_n) - 1)}_{\text{relative error}}}_{\text{absolute error}}
\end{aligned}$$

1.6.2 IEEE Floating Point Standard

Discussions regarding a standard for floating point arithmetic began around 1979 and was adopted by the IEEE in 1985. Most computer manufacturers and designers now adhere to this standard.

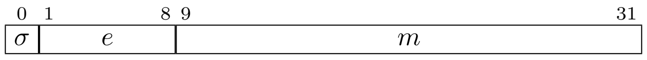
Basically the standard precisely defines two main formats — single and double precision floating point, and gives somewhat looser specifications for two extended versions of these formats.

The standard mandates that the operations $+, -, \times, \div$ and $\surd$ be provided, implemented to unit roundoff accuracy.

The extended formats are designed to be able to support the implementation of the operations to this level of accuracy.

IEEE allows four types of rounding: to $+\infty$, to $-\infty$, to 0, and to nearest (which is "rounding to even").

The format of a single precision IEEE floating point number is:



where σ denotes the sign ($\pm$), and values of e and m are interpreted as follows:

e	m	value
0	0	± 0
0	> 0	$0.m \cdot 2^{-126}$
$1 \ldots 254$	–	$1.m \cdot 2^{e-127}$
255	0	$\pm\infty$
255	> 0	NaN

NaN — Not a Number — used to flag errors such as divide by zero or overflow.

The double precision format is similar, but uses 11 bits for e and 52 bits for m.

1.7 3BA1-II Lecture 7

1.7.1 Function Evaluation

We consider techniques for evaluating functions such as polynomials, sin, cos, tan, etc.

First we consider polynomials of degree n:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x^1 + a_0 x^0$$

The obvious algorithm to evaluate this given an array a indexed from 0 to n is:

```
sum := 0;
for i := 0 to n do
  sum := sum + a[i] * power(x,i)
endfor
```

This involves n additions and $\sum_{k=0}^{n} k$ multiplies (assuming `power(x,i)` requires $i-1$ multiplications), for a total of floating point operations.

Horner's Rule

We compute the polynomial much more efficiently, using n adds, and only n multiplies, using "Horner's Rule,,:

$$(((\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots)x + a_2)x + a_1)x + a_0$$

or as an algorithm:

```
sum := a[n];
for i := n-1 downto 0 do
  sum := x * sum + a[i]
endfor
```

Truncation Error

Polynomials of finite degree are straightforward, because they are finite.

However many other functions such as sin, cos, etc, can be shown to be equivalent to polynomials of infinite degree, e.g.:

$$\sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k x^{(2k+1)}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

In order to compute using these series, we must stop at some point, and hope that the remaining terms are insignificant. We need some way to estimate the error caused by using such a finite approximation — this error is known as the "Truncation Error".

Consider summing an infinite series:

$$S = \sum_{k=0}^{\infty} x_k$$

and the result if we truncate at $k = N$:

$$S_N = \sum_{k=0}^{N} x_k$$

Then the truncation error (remainder) at N (R_N)is defined as

$$R_N = S - S_N = \sum_{k=N+1}^{\infty} x_k$$

What we want are methods to estimate R_N, given suitable knowledge about the series $S = x_0 + x_1 + x_2 + \cdots$.

1.7.2 Alternating Series

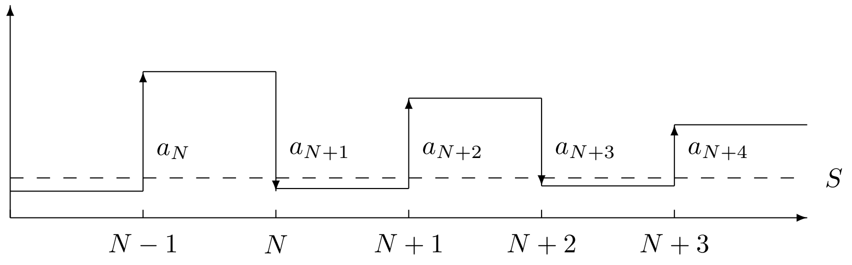
Consider a series where each successive terms has opposite sign and is smaller in magnitude than its predecessor:

$$S = a_1 - a_2 + a_3 - a_4 + a_5 + \cdots \qquad |a_n| > |a_{n+1}|$$

The sums are then:

$$\begin{aligned} S_1 &= a_1 \\ S_2 &= a_1 - a_2 \\ S_3 &= a_1 - a_2 + a_3 \\ &\vdots \end{aligned}$$

If we plot S_i against i we get the following, where we see that the sums converge towards the limit S:



We see that if we stop at S_N, having just added a_N, then that the error is bounded by the next term a_{N+1}, as all successive terms simply move us closer to the limit S.

So for such an alternating series we can conclude:

$$R_N = S - S_N \qquad |R_N| \leq |a_{N+1}|$$

The error bound is the size of the first term we ignore.

Example

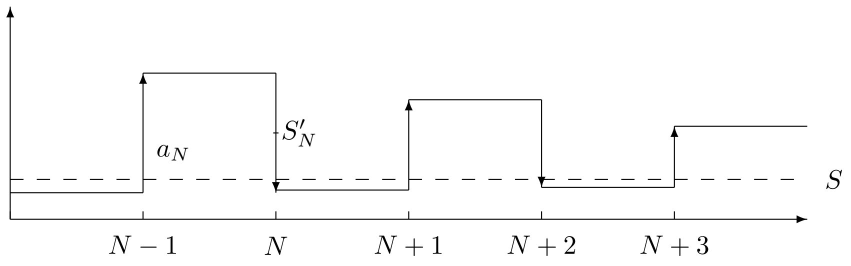
Compute $\sum \frac{(-1)^{n+1}}{n^2}$ accurate to 3 decimal places.

We want $|R_N| \leq \pm 0.5 \cdot 10^{-3}$, so we calculate

$$\begin{aligned} |R_N| &\leq |a_{N+1}| \\ &\leq \frac{1}{(N+1)^2} \\ &\leq \pm 0.5 \cdot 10^{-3} \end{aligned}$$

We solve this to get $N \geq \sqrt{2000} - 1 \approx 43.7$, so we need 44 iterations.

If we modify our algorithm to compute $S'_N = S_N + \frac{1}{2}a_{N+1}$ as our approximation, then the error is reduced by half:



So we get $|R'_N| = |S - S'_N| \leq \frac{1}{2}a_{N+1}$ If we solve this for $|R'_N| \leq \pm 0.5 \cdot 10^{-3}$ we get:$N \geq \sqrt{1000} - 1 \approx 30.6$, so only 31 iterations are required.

1.7.3 Bounded Monotonic Sequences

Consider a series whose elements are all positive in magnitude, but are strictly decreasing:

$$S = \sum a_n \qquad a_n \geq 0 \qquad a_n > a_{n+1}$$

If this series converges, then a_n tend to zero as n goes to infinity.

How can we estimate the error bound in this case.

If we can express the a_n as functions of n, we have a possibility. Consider a function f from reals to reals such that $f(n) = a_n$.

Then, the series sum is equivalent to summing the areas of blocks of width 1 and height a_n:

$$S = \sum_{)}^{\infty} a_n = \sum_{0}^{\infty} f(n)\Delta x \textbf{ where } \Delta x = 1$$

This is bounded from above by the corresponding integral of f:

$$S \leq \int_0^\infty f(x)dx$$

If we sum the first N terms then the remainder term R_N corresponds to

$$R_N = \sum_{N+1}^{\infty} a_n \leq \int_{N+1}^{\infty} f(x)dx = -F(N+1)$$

where F is the integral of f.

Example

Estimate the error in summing $\sum \frac{1}{n^4}$ to N terms.

$$\begin{aligned} R_N &\leq \int_{N+1}^{\infty} \frac{dx}{x^4} \\ &\leq \left.\frac{-3}{x^3}\right|_{N+1}^{\infty} \\ &\leq \frac{3}{(N+1)^3} \end{aligned}$$

So if we sum 9 terms, the error bound is

$$\frac{3}{(9+1)^3} = \frac{3}{10^3} = 3 \cdot 10^{-3}$$

For 99 terms, we get error bound $3 \cdot 10^{-6}$

1.8 3BA1-II Lecture 8

1.8.1 An efficient implementation of $\surd$

Or goal is to calculate square roots both efficiently and accurately !

How accurate: to the unit roundoff level, which for IEEE single precision is 2^{-24} (or approx. $6 \cdot 10^{-8}$).

The key algorithm we use is Newton-Raphson: For $\sqrt{a}$ we start with an an initial guess (x_0) and then iterate according to the following formula:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

The trick to finding an efficient answer is to get a good initial guess x_0.

Range Reduction

We shall exploit our knowledge about the underlying representation, in this case IEEE floating point using binary digits. Consider finding the square-root of A where

$$A = 1.b_1 b_2 \ldots b_{t-1} b_t \cdot 2^e$$

Nothing that $\sqrt{2^{2n}} = 2^n$, we realise that it would be convenient if the exponent was even ($e = 2k$). If the exponent is odd, however, we can make it even by subtracting one from it, and shifting the mantissa to the left to compensate. If the exponent is even, we leave it alone, but add a zero to the left of the mantissa, so that in each case we have a binary number with 2 digits to the left of the point:

e originally even	$e = 2k$	$A = 01.b_1 b_2 \ldots b_{t-1} b_t \cdot 2^{2k}$
e orignally odd	$e = 2k+1$	$A = 1b_1.b_2 \ldots b_{t-1} b_t 0 \cdot 2^{2k}$

In either case we have A of the form $a \cdot 2^{2k}$ where a is a bit-string of length $t+2$ with 2 digits to the left of the point. So now we can reason that

$$\sqrt{A} = \sqrt{a \cdot 2^{2k}} = \sqrt{a} \cdot \sqrt{2^{2k}} = \sqrt{a} \cdot 2^k$$

So the process of getting the root of the exponent simply involved dividing by two, which in binary is a simple shift right. All that remains is to find the root of a, which can in range in value from

$$01.00 \ldots 00$$

to

$$11.11 \ldots 10$$

In essence the general problem of finding the square root of an arbitrary number has been reduced to finding the root of a number between 1 and 4:

$$1 \leq a < 4$$

This process of reducing the range of values for which we have to actually compute the function under consideration is called **Range Reduction**. It is a very commonly used technique in numerical computation of functions. As another example, consider computing $\sin(x)$ for arbitrary x. As sin is a periodic function we know that

$$\sin(x) = \sin(x \pm 2\pi) = \sin(x \pm 4\pi) = \cdots = \sin(x \pm 2n\pi)$$

So, given arbitrary x we simply add or subtract multiples of 2π until we get a value in thew range $0 \ldots 2\pi$, and compute the sin of that.

Generating a good guess

We now have $1 \leq a < 4$, which means that the answer must be constrained to $1 \leq \sqrt{a} < 2$. This seems to give us good scope for an initial guess. However, we very quickly get an even better guess, by using some of the most significant bits of a to lookup a table of exact (pre-computed) solutions for those values. These then provide an initial guess very close to the desired result.

If we use four bits we get a table with 12 entries ($01.00 \ldots 11.11$):

$$\begin{array}{ccc} 01.00 & \mapsto & \sqrt{1.00} \\ 01.01 & \mapsto & \sqrt{1.25} \\ \vdots & \mapsto & \\ 11.10 & \mapsto & \sqrt{3.50} \\ 11.11 & \mapsto & \sqrt{3.75} \end{array}$$

Given that the interval between these entries is 2^{-2}, we can say our initial guess is accurate to this level (this is in fact quite conservative — a more detailed analysis using the mean value theorem gives an error of size 2^{-4} — remember the above value is the error in a, whereas what matters is the error in $\sqrt{a}$, which will be smaller).

Note that table lookup can be implemented very rapidly in hardware, and as an array lookup in software (multiply by 4, drop fraction, to get index between 4 and 15).

Applying Newton-Raphson

With our initial guess x_0 obtained from the lookup table, to accuracy 2^{w-2} (where w is no of bits used to index the table), we can now proceed to use Newton-Raphson.

We would like to estimate how many iterations we need to reach the desired accuracy. The error at each step is

$$\left| x_n - \sqrt{a} \right|$$

How does this behave as n grows ?

Analysis

We initially examine a tougher question — how do we know that we get the right answer at all ?

The following theorem gives our result:

Theorem: *For any x_0 such that $0 < x_0 < \infty$, we find (using Newton-Raphson) that*

$$x_1 \geq x_2 \geq x_3 \geq \cdots \geq x_n \geq x_{n+1} \geq \sqrt{a}$$

In other words all the values we compute descend from above down towards the root, and so converge to it.

Proof: First, assuming that $x_n > \sqrt{a}$ we derive an equation for $x_{n+1} - \sqrt{a}$:

$$\begin{array}{ll}
 & x_{n+1} - \sqrt{a} \\
= & \quad \text{“ defn. } x_{n+1} \text{ ”} \\
 & \frac{1}{2}\left(x_n - \frac{a}{x_n}\right) - \sqrt{a} \\
= & \quad \text{“ place all over } 2x_n \text{ ”} \\
 & \frac{x_n^2 - 2a - 2x_n\sqrt{a}}{2x_n} \\
= & \quad \text{“ rearrange ”} \\
 & \frac{x_n^2 - 2x_n\sqrt{a} - 2a}{2x_n} \\
= & \quad \text{“ } (z-b)^2 = z^2 - 2zb + b^2 \text{ with } z = x_n \text{ and } b = \sqrt{a} \text{ ”} \\
 & \frac{1}{2x_n}(x_n - \sqrt{a})^2 \\
\geq & \quad \text{“ } x_n \text{ positive implies expression is non-zero and positive ”} \\
 & 0 \\
\Rightarrow & \quad \text{“ } x - y \geq 0 \text{ means that } x \geq y \text{ ”} \\
 & x_{n+1} \geq \sqrt{a}
\end{array}$$

We now show that the sequence is decreasing:

$$\begin{array}{ll}
 & x_n - x_{n+1} \\
= & \quad \text{“ defn. } x_{n+1} \text{ ”} \\
 & x_n - \dfrac{1}{2}\left(x_n - \dfrac{a}{x_n}\right) \\
= & \quad \text{“ place all over } 2x_n \text{ ”} \\
 & \dfrac{2x_n^2 - x_n^2 - a}{2x_n} \\
= & \quad \text{“ simplify ”} \\
 & \dfrac{1}{2}(x_n^2 - a) \\
\geq & \quad \text{“ } x_n > \sqrt{a} \text{ means that } x_n^2 \geq a \text{ ”} \\
 & 0 \\
\Rightarrow & \quad \text{“ } x - y \geq 0 \text{ means that } x \geq y \text{ ”} \\
 & x_n \geq x_{n+1}
\end{array}$$

End of Proof

We can now use the expression for $x_{n+1} - \sqrt{a}$ to get an error estimate:

$$\left|x_{n+1} - \sqrt{a}\right| = \left|\frac{1}{2x_n}(x_n - \sqrt{a})^2\right|$$

Given our initial guess from our table is accurate to about 2^{-4}, we can say then that:

$$|x_0 - sqrta| \approx 2^{-4}$$

We can now substitute this into the equations for $n = 1, 2, 3, \ldots$, assuming as a worst case that all $x_i \approx 1$ (as this gives the largest error estimates:

$$\begin{array}{lcl}
\left|x_1 - \sqrt{a}\right| & \approx & \left|\dfrac{1}{2x_0}(x_0 - \sqrt{a})^2\right| \leq \left|\dfrac{1}{2}(2^{-4})^2\right| = \left|2^{-9}\right| \\
\left|x_2 - \sqrt{a}\right| & \approx & \left|\dfrac{1}{2x_1}(x_1 - \sqrt{a})^2\right| \leq \left|\dfrac{1}{2}(2^{-9})^2\right| = \left|2^{-19}\right| \\
\left|x_3 - \sqrt{a}\right| & \approx & \left|\dfrac{1}{2x_2}(x_2 - \sqrt{a})^2\right| \leq \left|\dfrac{1}{2}(2^{-19})^2\right| = \left|2^{-39}\right|
\end{array}$$

We see that after 3 iterations that the truncation error is less than 2^{-24}, the unit roundoff error, so three iterations will suffice.

If we use 6 bits to index the table, needing 48 entries, we only need two iterations to get the result.

1.9 3BA1-II Lecture 9

1.9.1 Root Finding

We are trying to find an x such that

$$f(x) = 0$$

for non-linear f, which we know how to compute.

We shall assume two things to make life simpler: that f is differentiable, and that that the root is **simple** ($f'(x) \neq 0$ at root).

We shall let x^* denote the root.

Technique:

1. Make a good guess (x_0)
2. Use iterative technique to improve the guess, generating series

$$x_0, x_1, x_2, \ldots, x_n, \ldots \to x^*$$

There are two broad classes of iterative techniques:

1. Intervals — we attempt to bracket the root in an interval and then shrink the interval around the root until the desired accuracy is reached.
2. Points — we take a single guess and try to move it closer to the root

We will find that most interval techniques are safe, but slow, while point techniques tend to be faster, but run the risk of diverging.

In the sequel, we shall use the following running example:

$$x - e^{-x} = 0$$

1.9.2 Generating initial guesses

One technique is to plot the graph of the function. In this case, we note that we are in fact solving $x = e^{-x}$, and so it is easiest to plot e^{-x} and x and to see where they intersect.

Another technique is to tabulate values and look for a change in sign (this technique is easier to automate than the graphing one).

If we do that for our example, we will discover that a root lies between 0.5 and 0.6:

For $x - e^{-x}$, we have $0.5 < x^* < 0.6$.

1.9.3 Interval Techniques

Bisection

Given an initial interval, we determine subsequent intervals by finding the mid-point, evaluating f at that point, and then producing a new interval with the mid-point as one end, and the original end-point where f has a different sign at the other end.

Given interval $[x_{n-1}, x_n], \mathsf{sign}(f(x_{n-1})) \neq \mathsf{sign}(f(x_n))$ we compute

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

If $\mathsf{sign}(f(x_{n+1})) = \mathsf{sign}(f(x_n))$ then the new interval is $[x_{n-1}, x_{n+1}]$, otherwise it is $[x_n, x_{n+1}]$.

This technique is robust — if the starting interval encloses the root, then all subsequent intervals will.

It is slow — the interval size (accuracy of result) halves at each iteration.

Consider the starting interval $[0.5, 0.6]$, of size 10^{-1}, or about 2^{-3}. To get to an interval of width approximately 10^{-6} (IEEE Single precision) we need about 17 iterations.

As a general principle, we hope to find that we can get faster algorithms, by making more use of the information we have about f. The bisection technique only makes use of the sign of $f(x)$.

Regula Falsi

The **Regula Falsi** technique exploits knowledge about the approximate slope of the function around the root to improve the next guess.

Given interval $[x_{n-1}, x_n], \mathsf{sign}(f(x_{n-1})) \neq \mathsf{sign}(f(x_n))$, we can determine two points as $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$. We determine the equation of the straight-line between them as :

$$\frac{y - f(x_n)}{x - x_n} = \frac{f(x_{n-1}) - f(x_n)}{x_{n-1} - x_n}$$

We shall let our next interval end-point ($x = x_{n+1}$) be where this line intersects the x-axis ($y = 0$). If we make these substitutions we get:

$$\frac{0 - f(x_n)}{x_{n+1} - x_n} = \frac{f(x_{n-1}) - f(x_n)}{x_{n-1} - x_n}$$

We multiply both sides by the reciprocal of the righthand side to get:

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

We multiply both sides by $x_{n+1} - x_n$ to get:

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

and a final re-arrangement gives:

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

or equivalently:

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

Once we have computed x_{n+1} in this way, we determine the new interval exactly as we did for the bisection method.

This method is robust, but is still slow.

Secant

The secant method uses the same formula as Regula Falsi, but it does not attempt to keep the interval surrounding the root. Instead the new interval is $[x_n, x_{n+1}]$, regardless of the signs of the functions at those points.

The secant method converges faster than Regula Falsi, when it converges at all. It is not robust.

A key idea is to use the robust (but slow) techniques to improve initial guesses to the point were the faster, less robust tehniques can be safely used. Usually this point occurs where we are close enough to the root that the function is "well-behaved" with no great changes in value or slope.

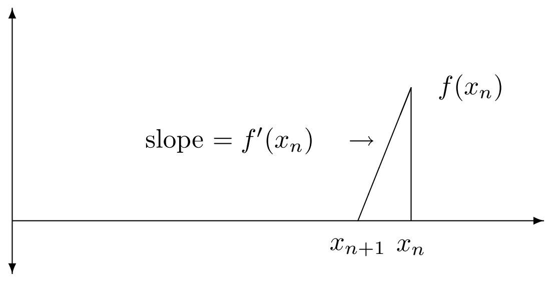
1.9.4 Point Techniques

Point techniques don't use intervals — rather we have a single point as our initial guess, and we repeatedly make it better (we hope !).

Newton-Raphson

An example of a point method is the **Newton-Raphson** technique: we use the slope and value of f at our guess to provide us with a better guess.

We take the point $(x_n, f(x_n))$, and determine the tangent line at that point to find its intersection with the x-axis.



The slope of the line is

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

where $f(x_{n+1}) = 0$ and this slope equals $f'(x_n)$. So we get

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

Re-arranging gives:

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

This is the iteration formula for Newton-Raphson.

This method is very fast, and quickly reaches a solution, if it converges. However it is not completely robust.

When it does converge, it is very fast, typically doubling the number of significant digits in the answer with each iteration.

Fixed Point Iteration

Another approach is to compute so-called fixed-point solutions.

We take an equation of the form $f(x) = 0$ and transform it into one of the form $\varphi(x) = x$, where φ is related to f in such a way that $f(x^*) = 0$ iff $\varphi(x^*) = x^*$.

First example, consider

$$\varphi_1(x) = e^{-x}$$

So we are solving $x = e^{-x}$, which as we have already observed, is equivalent to solving $x - e^{-x} = 0$.

We simply start with our initial guess x_0 and repeatedly apply φ:

$$x_{n+1} = \varphi(x_n)$$

We generate a series we hope converges to the solution x^*:

$$x_0, \varphi(x_0), \varphi(\varphi(x_0)), \varphi^3(x_0), \varphi^4(x_0), \ldots \rightarrow x^*$$

If we try this with $x_0 = 0.5$ we see that it converges to the required result, but quite slowly.

As another example, consider

$$\varphi_2(x) = -logx$$

We show that the solution to $-logx = x$ is the same as that for $x - e^{-x} = 0$ by plugging this value in: it into the equation for f:

$$\begin{aligned} f(-logx) &= -logx - e^{-(-logx)} \\ &= -logx - e^{logx} \\ &= -logx - x \end{aligned}$$

If x^* is a solution to $-logx - x = 0$, then it means that $x^* = -logx^*$, and it is a solution to $f(-logx) = 0$ as we have just shown, so

$$f(-logx^*) = 0 = f(x^*) \qquad \textbf{and} \qquad x^* = -logx^*$$

However, if we try this fixed-point equation with $x_0 = 0.5$, we find that the values rapidly diverge.

Finally, we note that Newton-Raphson can be formulated as a a fixed point iteration: assume that $f(x^*) = 0$. Then, if

$$\varphi_3(x) = x - \frac{f(x)}{f'(x)}$$

we see that:

$$\begin{aligned} & \varphi_3(x^*) \\ = & \quad \text{“ defn. } \varphi_3 \text{ ”} \\ & x^* - \frac{f(x^*)}{f'(x^*)} \\ = & \quad \text{“ } x^* \text{ is a root of } f\text{, root is simple, so } f'(x^*) \neq 0 \text{ ”} \\ & x^* - \frac{0}{f'(x^*)} \\ = & \quad \text{“ clean up ”} \\ & x^* \end{aligned}$$

We have shown that $\varphi_3(x^*) = x^*$, i.e. that it is a fixed point of φ_3.

Is there a general theory that predicts if and how fast these fixed-points converge ?

1.10 3BA1-II Lecture 10

1.10.1 Analysis of Iterative Methods

The point-based method of finding roots by solving fixed-point equations is amenable to analysis in order to establish criteria for convergence.

We are finding solutions to $f(x) = 0$, by solving $\varphi(x) = x$ for ϕ related appropriately to f. The idea is that the solution (x^*) implies that

$$f(x^*) = 0 \qquad \Leftrightarrow \qquad \varphi(x^*) = x^*$$

The example we worked with was

$$f(x) = x - e^{-x}$$

and experimentation with initial guess $x_0 = 0.5$ and three different versions of φ gave the following outcomes:

φ	equation	outcome
$\varphi_1(x) = e^{-x}$	$x = e^{-x}$	OK, slow
$\varphi_2(x) = -logx$	$x = -logx$	NOT OK, diverges
$\varphi_3(x) = x - \frac{f(x)}{f'(x)}$		OK, FAST

Is there a theory to account for these observations ?

1.10.2 Convergence Analysis

We start with the basic fixed-point iteration step:

$$x_{n+1} = \varphi(x_n)$$

We define the error at the nth iteration (ϵ_n) as:

$$\epsilon_n = x_n - x^*$$

We observe that $x_n \to x^*$ if and only if $\epsilon_n \to 0$, as $n \to \infty$.

We now develop the expression for ϵ_n:

$$\begin{array}{ll}
 & \epsilon_n \\
= & \quad \text{“ defn. } \epsilon_n \text{ ”} \\
 & x_n - x^* \\
= & \quad \text{“ iteration step ”} \\
 & \varphi(x_{n-1})) - x^* \\
= & \quad \text{“ } x^* \text{ is a fixed-point of } \varphi \text{ ”} \\
 & \varphi(x_{n-1})) - \varphi(x^*) \\
= & \quad \text{“ Mean Value Theorem, for } \xi \text{ between } x_{n-1} \text{ and } x^* \text{ ”} \\
 & \varphi'(\xi)(x_{n-1} - x^*) \\
= & \quad \text{“ defn. } \epsilon_{n-1} \text{ ”} \\
 & \varphi'(\xi)\epsilon_{n-1}
\end{array}$$

We have shown that

$$\boxed{\epsilon_n = \varphi'(\xi)\epsilon_{n-1}}$$

In order for ϵ_n to be smaller than ϵ_{n-1}, we impose the following convergence criteria, that there exists m such that:

$$|\varphi'(\xi)| \leq m < 1$$

for all ξ close to x^*.

If this is the case then we can conclude:

$$\begin{array}{rcl}
|\epsilon_n| & \leq & |\varphi'(\xi)|\,|\epsilon_{n-1}| \\
 & \leq & m\,|\epsilon_{n-1}| \\
 & \leq & m^2\,|\epsilon_{n-2}| \\
 & \leq & \cdots \\
 & \leq & m^{n-1}\,|\epsilon_1| \\
 & \leq & m^n\,|\epsilon_0|
\end{array}$$

So we have

$$|\epsilon_n| \leq m^n\,|\epsilon_0|$$

and $m^n \to 0$ as $n \to \infty$, so we can conclude that $\epsilon_n \to 0$.

The quantity $\varphi'(\xi)$ is the slope of φ close to x^*. If this slope's magnitude is less tnan one, we converge to the fixed point.

If we look at our first two examples, we we take $\xi = 0.567$ (close to the root):

$$\begin{array}{lll}
\varphi_1(x) = e^{-x} & \varphi_1'(x) = -e^{-x} & |\varphi_1'(0.567)| \approx 0.567 < 1 \\
\varphi_2(x) = -log x & \varphi_2'(x) = -1/x & |\varphi_2'(0.567)| \approx 1/0.567 > 1
\end{array}$$

The convergence criteria matches our experimental observations.

Note that the convergence described here is **linear** — the error is reduced by a constant factor times itself at each iteration.

Now we turn our attention to Newton-Raphson:

$$\begin{aligned}\varphi_3(x) &= x - \frac{f(x)}{f'(x)} \\ \varphi_3'(x) &= \frac{f(x)f''(x)}{(f'(x))^2}\end{aligned}$$

The derivative of φ_3 is calculated as follows:

$$\begin{array}{ll} & \varphi_3'(x) \\ = & \quad \text{“ defn. } \varphi_3 \text{ ”} \\ & \dfrac{d(x - f(x)/f'(x))}{dx} \\ = & \quad \text{“ laws of differentiation ”} \\ & 1 - \dfrac{d(f(x)/f'(x))}{dx} \\ = & \quad \text{“ } d\dfrac{u}{v} = \dfrac{vdu - udv}{v^2} \text{ ”} \\ & 1 - \dfrac{(f'(x))^2 - f(x)f''(x)}{f'(x)f'(x)} \\ = & \quad \text{“ algebra ”} \\ & 1 - \dfrac{(f'(x))^2}{(f'(x))^2} + \dfrac{f(x)f''(x)}{(f'(x))^2} \\ = & \quad \text{“ simplify ”} \\ & 1 - 1 + \dfrac{f(x)f''(x)}{(f'(x))^2} \\ = & \quad \text{“ simplify ”} \\ & \dfrac{f(x)f''(x)}{(f'(x))^2} \end{array}$$

We compute at the fixed point:

$$\begin{array}{ll} & \varphi_3'(x^*) \\ = & \quad \text{“ defn. } \varphi_3' \text{ ”} \\ & \dfrac{f(x^*)f''(x^*)}{(f'(x^*))^2} \\ = & \quad \text{“ Root is simple } (f'(x^*) \neq 0 \text{, and } f(x^*) = 0 \text{ ”} \\ & 0 \end{array}$$

So $\varphi_3'(\xi)$ (ξ near (x^*)) is very small, so we get rapid convergence, as observed.

Our method of convergence analysis up to this point suggests to us that m is close to zero for Newton-Raphson. Can we get a more precise estimate than “close to zero” ?

1.10.3 Detailed Analysis of Newton-Raphson

We have already established that

$$\epsilon_{n+1} = x_{n+1} - x^* = \varphi(x_n) - \varphi(x^*)$$

So we start with righthand-most expression:

$$\begin{array}{ll}
 & \varphi(x_n) - \varphi(x^*) \\
= & \quad \text{“ Taylor's Law, expanding around } x^*\text{, for } \xi \text{ between } x_n \text{ and } x^* \text{ ”} \\
 & \varphi'(x^*)(x_n - x^*) + \dfrac{\varphi''(\xi)}{2!}(x_n - x^*)^2 \\
= & \quad \text{“ } \varphi'(x^*) = 0\text{, as already shown ”} \\
 & \dfrac{\varphi''(\xi)}{2!}(x_n - x^*)^2 \\
= & \quad \text{“ defn. } \epsilon_n \text{ ”} \\
 & \dfrac{\varphi''(\xi)}{2!}\epsilon_n^2
\end{array}$$

We have shown that, for Newton-Raphson:

$$\boxed{\epsilon_{n+1} = \varphi''(\xi)\epsilon_n^2}$$

for some ξ close to the fixed point.

The key feature to note is that the error reduces **quadratically**, with the next error being a constant time the previous error *squared.*

$$|\epsilon_{n+1}| \leq k\,|\epsilon_n|$$

This justifies the statement made previously that typically (i.e when k is not too large) Newton-Raphson doubles the number of significant digits per iteration. It also explains why Newton-Raphson converges faster than general fixed point iterations.

1.10.4 Finding Roots - Summary

So overall, the best technique for finding roots, in most cases, is to make an good initial guess (graphing, tabulating), then use Regula Falsi or bisection to narrow down the interval to a well-behaved section around the root, and then apply Newton-Raphson to finish of quickly and accurately.

1.11 3BA1-II Lecture 11

1.11.1 Numerical Differentiation

We assume that we can compute a function f, but that we have no information about how to compute f'. We want ways of estimating $f'(x)$, given what we know about f.

Reminder: definition of differentiation:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

We can use this formula, by taking Δx equal to some small value h, to get the following approximation, known as the **Forward Difference** ($D_+(h)$):

$$f'(x) \approx D_+(h) = \frac{f(x+h) - f(x)}{h}$$

Alternatively we could use the interval on the other side of x, to get the Backward Difference ($D_-(h)$) :

$$f'(x) \approx D_-(h) = \frac{f(x) - f(x-h)}{h}$$

A more symmetric form, the Central Difference ($D_0(h)$), uses intervals on either side of x:

$$f'(x) \approx D_0(h) = \frac{f(x+h) - f(x-h)}{2h}$$

All of these give (different) approximations to $f'(x)$.

For second derivatives, we have the definition:

$$\frac{d^2 f}{dx^2} = \lim_{\Delta x \to 0} \frac{f'(x + \Delta x) - f'(x)}{\Delta x}$$

The simplest way is to get a symmetrical equation about x by using both the forward and backward differences to estimate $f'(x+\Delta x)$ and $f'(x)$ respectively:

$$f''(x) \approx \frac{D_+(h) - D_-(h)}{h} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

1.11.2 Error Estimation

We shall see that the error involved in using these differences is a form of truncation error (R_T).

We first look at the error associated with forward difference:

$$
\begin{array}{ll}
 & R_T \\
= & \text{“ difference between estimate and true value ”} \\
 & D_+(h) - f'(x) \\
= & \text{“ defn. } D_+(h) \text{ ”} \\
 & \frac{1}{h}(f(x+h) - f(x)) - f'(x) \\
= & \text{“ Taylor's Theorem: } f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + f^{(3)}(x)h^3/3! + \cdots \text{ ”} \\
 & \frac{1}{h}(f'(x)h + f''(x)h^2/2! + f'''(x)h^3/3! + \cdots) - f'(x) \\
= & \text{“ algebra ”} \\
 & \frac{1}{h}f'(x)h + \frac{1}{h}(f''(x)h^2/2! + f'''(x)h^3/3! + \cdots)) - f'(x) \\
= & \text{“ more algebra ”} \\
 & f''(x)h/2! + f'''(x)h^2/3! + \cdots \\
= & \text{“ Mean Value Theorem, for some } \xi \text{within } h \text{ of } x \text{ ”} \\
 & \frac{1}{2}f''(\xi)h
\end{array}
$$

We don't know the value of either f'' or ξ, but we can say that the error is order h:

$$R_T \text{ for } D_+(h) \text{ is } O(h)$$

so the error is proportional to the step size — as one might naively expect.
For $D_-(h)$ we get a similar result for the truncation error — also $O(h)$.

1.11.3 Exercise

The rest of this lecture was then given over to an explanation of Exercise 1.

1.12 Exercise (2004) No. 1

1.12.1 Objective

To numerically solve the diffusion equation (from 3BA4) to verify if propagation time down long thin wires really is $O(l^2)$.

We are solving the (partial) differential equation

$$rc\frac{\partial V}{\partial t} = \frac{\partial^2 V}{\partial x^2}$$

for $V(x,t)$ as a function of both space ($x \geq 0$) and time ($t \geq 0$), subject to the following initial and boundary conditions:

$$\begin{array}{rcll} V(x,0) & = & 0V, & x > 0 \\ V(0,t) & = & 5V, & t \geq 0 \end{array}$$

1.12.2 Methodology

We represent the voltage at different locations (x) at a given time t by an array of 21 elements (voltages at locations $x_0, x_1, \ldots, x_{20}$).

We use the following recurrence formula, given an array of voltages at time t, to compute the voltages at time $t+1$:

$$V(x,t+1) = V(x,t) + k \cdot (V(x+1,t) - 2V(x,t) + V(x-1,t))$$

where

$$k = \frac{\Delta t}{rc\Delta x^2}$$

We initialise the array such that the 0th entry has value 5.0, with the rest set to zero.

In future time-step, the 0th value never changes — i.e. is fixed permanently.

The idea is to iterate over a number of time steps, to get data about voltages at each location during each time-step.

To explore if propagation time is $O(l^2)$, you should pick a threshold value[1] between 0.0 and 5.0, and for each location $x_1, x_2, \ldots, x_{20}$, and record at what iteration (time-step) the voltage at that point exceed the threshold for the first time.

Programming

You must solve this problem using a computer, either by programming the solution yourself (in your favourite language), or by using a spreadsheet.

[1]3BA4 talked about a threshold of 90% of the maximum, but any threshold will do for this experiment — you may find that you have to iterate quite a long time before the end voltage reached $4.5V$ in this study

1.12.3 Questions

You should run enough experiments and amass enough data to answer the following questions:

1. What value of k is best for getting out the results ?
2. Are any values of k bad ?
3. Does you data corroborate the hypothesis that the time for a signal to propagate is $O(l^2)$?

Your answer should not be a simple yes/no or a number, but should contain an explanation of why you believe your answer to be correct.

1.12.4 Submission

All your files (program, results, text commentary) should be bundled into a single archive (WinZip, compress, tar) with your username as filename, and emailed to `Andrew.Butterfield@cs.tcd.ie` with the subject line "3BA1-Diffusion" (exactly as spelt here), by 9am, Thursday, April 29th.

1.13 3BA1-II Lecture 12

1.13.1 Error analysis of Central Difference

We consider the error in the Central Difference estimate ($D_0(h)$) of $f'(x)$:

$$D_0(h) = \frac{f(x+h) - f(x-h)}{2h}$$

We apply Taylor's Theorem,

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)h^2}{2!} + \cdots + \frac{f^{(n)}(x)h^n}{n!} + \cdots$$

creatively:

$$\begin{array}{rcll}
f(x+h) & = & f(x) + f'(x)h + \frac{f''(x)h^2}{2!} + \frac{f'''(x)h^3}{3!} + \frac{f^{(4)}(x)h^4}{4!} + \cdots & (A) \\
f(x-h) & = & f(x) - f'(x)h + \frac{f''(x)h^2}{2!} - \frac{f'''(x)h^3}{3!} + \frac{f^{(4)}(x)h^4}{4!} + \cdots & (B) \\
(A) - (B) & = & 2f'(x)h + 2\frac{f'''(x)h^3}{3!} + 2\frac{f^{(5)}(x)h^5}{5!} + \cdots & \\
\frac{(A) - (B)}{2h} & = & f'(x) + \frac{f'''(x)h^2}{3!} + \frac{f^{(5)}(x)h^4}{5!} + \cdots &
\end{array}$$

We see that the difference can be written as

$$D_0(h) = f'(x) + \frac{f''(x)}{6}h^2 + \frac{f^{(4)}(x)}{24} + \cdots$$

or alternatively, as

$$D_0(h) = f'(x) + b_1 h^2 + b\_2h^4 + \cdots$$

where be know how to compute b_1, b_2, etc. We see that the error ($R_T = D_0(h) - f'(x)$) is $O(h^2)$.
Does this theory work ? Let us try an example:

$$f(x) = e^x \qquad f'(x) = e^x$$

We evaluate $f'(1) = e^1 \approx 2.71828...$, starting with $h = 0.4$, and halving the interval each time.

$$D_0(h) = \frac{f(1+h) - f(1-h)}{2h}$$

h	$D_0(h)$	$D_0(h) - e$
0.4	2.791352	$7.31 \cdot 10^{-2}$
0.2	2.736440	$1.82 \cdot 10^{-2}$
0.1	2.722815	$4.53 \cdot 10^{-3}$
0.05	2.719414	$1.13 \cdot 10^{-3}$

We see that as the value of h halves, that the error drops by about a quarter in each case. So, can we assume that

$$R_T \to 0 \text{ as } h \to 0 \qquad ?$$

Let us consider values of h closer to the unit-roundoff. If we perform this experiment on machine with unit roundoff 2^{-27} ($\approx 7.5 \cdot 10^{-9}$), using $h = 7.5 \cdot 10^{-x}$ for $x = 1, \ldots, 9$, we get:

h	$D_0(h)$	$D_0(h) - e$	
$7.5 \cdot 10^{-1}$	2.976847	2.59×10^{-1}	
$7.5 \cdot 10^{-2}$	2.720797	2.52×10^{-3}	
$7.5 \cdot 10^{-3}$	2.718306	2.42×10^{-5}	
$7.5 \cdot 10^{-4}$	2.718300	1.82×10^{-5}	!
$7.5 \cdot 10^{-5}$	2.718200	-8.18×10^{-5}	!!
$7.5 \cdot 10^{-6}$	2.718000	-2.82×10^{-4}	
$7.5 \cdot 10^{-7}$	2.720000	1.72×10^{-3}	
$7.5 \cdot 10^{-8}$	2.800000	8.17×10^{-2}	
$7.5 \cdot 10^{-9}$	4.000000	1.28	

We can see that the error shrinks by about a factor of 100, until $7.5 \cdot 10^{-4}$(!), where the shrinkage is is only about half. It gets worse at the next drop in h, because the error actually gets larger (!!), and this gets worse as h shrinks, until we see the error for the smallest h is almost 50% of the true value.

1.13.2 Rounding Error in Difference Equations

When presenting the iterative techniques for root-finding, we ignored rounding errors, and paid no attention to the potential error problems with performing subtraction. This did not matter for such techniques because:

1. the techniques are "self-correcting", and tend to cancel out the accumulation of rounding errors
2. the iterative equation $x_{n+1} = x_n - c_n$ where c_n is some form of "correction" factor[2] has a subtraction which is safe because we are subtracting a small quantity (c_n) from a large one.

However, when using a difference equation like

$$D_0(h) = \frac{f(x+h) - f(x-h)}{2h}$$

we seek a situation where h is small compared to everything else, in order to get a good approximation to the derivative. This means that $x + h$ and $x - h$ are very similar in magnitude, and this means that for most f (well-behaved) that

[2]e.g. for Newton-Raphson, $c_n = \frac{f(x)}{f'(x)}$

$f(x+h)$ will be very close to $f(x-h)$. So we have the worst possible case for subtraction: the difference between two large quantities whose values are very similar.

We cannot "re-arrange" the equation to get rid of the subtraction, as this difference is inherent in what it means to compute an approximation to a derivative — differentiation uses the concept of difference in a deeply intrinsic way.

We see now that the total error in using $D_0(h)$ to estimate $f'(x)$ has two components — the truncation error R_T which we have already calculated, and a function calculation error R_{XF} which we now examine.

When calculating $D_0(h)$, we are not using totally accurate computations of f, but instead we actually compute an approximation $\bar{f}$, to get

$$\bar{D}_0(h) = \frac{\bar{f}(x+h) - \bar{f}(x-h)}{2h}$$

We shall assume that the error in computing f near to x is bounded in magnitude by ϵ:

$$\left|\bar{f}(x) - f(x)\right| \leq \epsilon$$

The calculation error is then given as

$$\begin{array}{rcl}
R_{XF} & = & \bar{D}_0(h) - D_0(h) \\
 & = & \quad \text{“ defn. } \bar{D}_0, D_0 \text{ ”} \\
 & & \dfrac{\bar{f}(x+h) - \bar{f}(x-h)}{2h} - \dfrac{f(x+h) - f(x-h)}{2h} \\
 & = & \quad \text{“ common denominator ”} \\
 & & \dfrac{\bar{f}(x+h) - \bar{f}(x-h) - (f(x+h) - f(x-h))}{2h} \\
 & = & \quad \text{“ re-arrange ”} \\
 & & \dfrac{\bar{f}(x+h) - f(x+h) - (\bar{f}(x-h) - f(x-h))}{2h} \\
 & & \quad \text{“ take magnitudes, triangle inequality ”} \\
|R_{XF}| & \leq & \dfrac{\left|\bar{f}(x+h) - f(x+h)\right| + \left|\bar{f}(x-h) - f(x-h)\right|}{2h} \\
 & \leq & \quad \text{“ error bounds for } f \text{ ”} \\
 & & \dfrac{\epsilon + \epsilon}{2h} \\
 & = & \quad \text{“ simplify ”} \\
 & & \dfrac{\epsilon}{h}
\end{array}$$

So we see that R_{XF} is proportional to $1/h$, so as h shrinks, this error grows, unlike R_T which shrinks quadratically as h does.

We see that the total error R is bounded by $|R_T| + |R_{XF}|$, which expands out to

$$|R| \leq \left|\frac{f'''(\xi)}{6}h^2\right| + \left|\frac{\epsilon}{h}\right|$$

So we see that to minimise the overall error we need to find the value of $h = h_{opt}$ which minimises the following expression:

$$\frac{f'''(\xi)}{6}h^2 + \frac{\epsilon}{h}$$

Unfortunately, we do not know f''' or ξ !

Many techniques exist to get a good estimate of h_{opt}, most of which estimate f''' numerically somehow. These are complex and not discussed here.

1.13.3 Richardson Extrapolation

We present a technique for improving our difference estimate by making use of the variation in estimates we get using different values of h, plus knowledge about the behaviour of truncation errors in order to improve our estimate.

The trick is to compute $D(h)$ for 2 different values of h, and combine the results in some appropriate manner, as guided by our knowledge of the error behaviour.

Task: Given $D(h)$ computable for $h \neq 0$, determine

$$\lim_{h \to 0} D(h)$$

If we know R_T as a function of h, we can use this to get a good approximation to the limit.

We shall let D be D_0 for this example (this works just as well for D_+ and D_-). In this case we have already established that

$$D_0(h) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + b_1 h^2 + O(h^4)$$

We now consider using twice the value of h:

$$D_0(2h) = \frac{f(x+2h) - f(x-2h)}{4h} = f'(x) + b_1 4h^2 + O(h^4)$$

We can subtract these to get:

$$D_0(2h) - D_0(h) = 3b_1 h^2 + O(h^4)$$

Note: the $O(h^4)$ terms do not disappear when subtracting as in each of the equations they denote unknown and almost certainly different values whose magnitude is of order h^4.

We divide across by 3 to get:

$$\frac{D_0(2h) - D_0(h)}{3} = 3b_1 h^2 + O(h^4)$$

The righthand side of this equation is simply $D_0(h) - f'(x)$, so we can substitute to get

$$\frac{D_0(2h) - D_0(h)}{3} = D_0(h) - f'(x)$$

This re-arranges (carefully) to obtain

$$f'(x) = D_0(h) + \frac{D_0(h) - D_0(2h)}{3} + O(h^4)$$

We see that

$$D_0(h) + \frac{D_0(h) - D_0(2h)}{3} \qquad = \qquad \frac{4D_0(h) - D_0(2h)}{3}$$

is an estimate for $f'(x)$ whose truncation error is $O(h^4)$, and so is an improvement over D_0 used alone.

This technique of using calculations with different h values to get a better estimate is known as **Richardson Extrapolation**.

An analysis of calculation error in this case also shows an improvement, if we iterate this technique in a certain fashion, until successive answers do not change. — we find in this case that the total error ends up being about 2ϵ, rather than ϵ/h.

1.14 3BA1-II Lecture 13

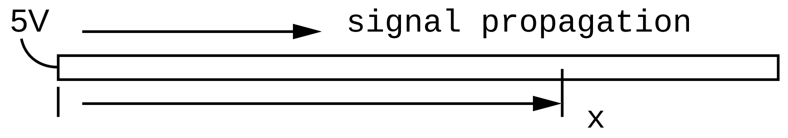
1.14.1 Discussion of Exercise (2004) No. 1 (§1.12)

Experimental Goal

What was required was a numerical solution to the Diffusion Equation, to try to verify the $O(\ell^2)$ propagation speed.

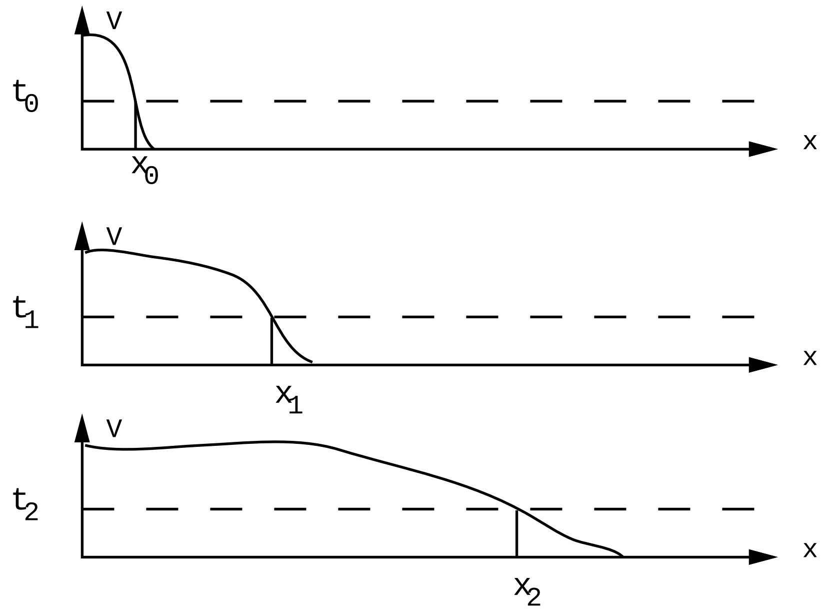
For any experiment, it is important to know in advance what outcome to expect.

For this experiment, we hope to see that a signal propagates along a long thin wire:



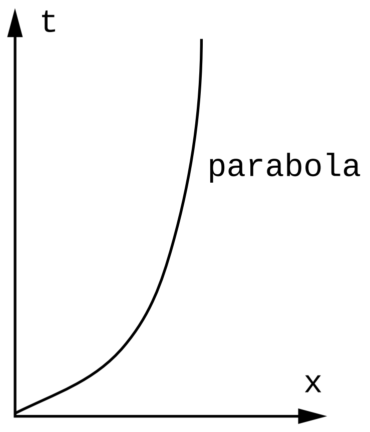
We are hoping to see that the time it takes for a voltage of a certain threshhold value to reach point x is proportional to x^2.

If $t_0 = 0 + \epsilon$ is a time very shortly after the signal starts propagating, and t_1 and t_2 are time intervals somewhat later the what we hope to see as time goes by is something like:

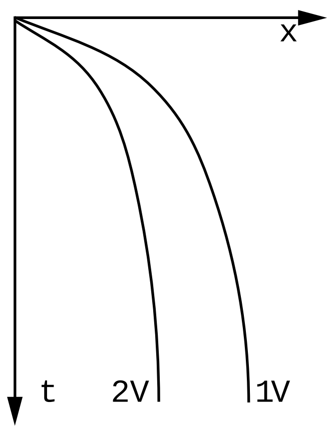


We hope to see that $t_i \propto x_i^2$ or conversely, that $x_i \propto \sqrt{t_i}$.

If we plot t vs x, we would expect to see something like plot (A) below:



(A)



(B)

The plot (B) shows the plot re-arranged so the axes' directions match they way most programs and spreadsheet solutions would show the numbers — we see also how the curves vary according to the threshold voltage chosen to generate the data.

Procedure

The procedure is very simple:

1. Generate the data, by running the program with a value of k, for some number of time-slots.
2. Analyse the data — for each time-slot, determine the index of the first location where the voltage is less than the threshold

The following shows an actual run of the experiment, limited to the first 10 x-locations along the wire, using $k = 0.5$, showing such first locations in bold, for a threshold of 0.5 with the corresponding index-data x_t:

t	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_t
0	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1
1	2.50	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2
2	2.50	1.25	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3
3	3.12	1.25	0.62	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00	4
4	3.12	1.87	0.62	**0.31**	0.00	0.00	0.00	0.00	0.00	0.00	4
5	3.43	1.87	1.09	**0.31**	0.15	0.00	0.00	0.00	0.00	0.00	4
6	3.43	2.26	1.09	0.62	**0.15**	0.07	0.00	0.00	0.00	0.00	5
7	3.63	2.26	1.44	0.62	**0.35**	0.07	0.03	0.00	0.00	0.00	5
8	3.63	2.53	1.44	0.89	**0.35**	0.19	0.03	0.01	0.00	0.00	5
9	3.76	2.53	1.71	0.89	0.54	**0.19**	0.10	0.01	0.01	0.00	6
10	3.76	2.74	1.71	1.13	0.54	**0.32**	0.10	0.05	0.01	0.00	6
11	3.87	2.74	1.93	1.13	0.72	**0.32**	0.19	0.05	0.03	0.00	6
12	3.87	2.90	1.93	1.33	0.72	**0.46**	0.19	0.11	0.03	0.01	6
13	3.95	2.90	2.11	1.33	0.89	**0.46**	0.28	0.11	0.06	0.01	6
14	3.95	3.03	2.11	1.50	0.89	0.59	**0.28**	0.17	0.06	0.03	7
15	4.01	3.03	2.27	1.50	1.05	0.59	**0.38**	0.17	0.10	0.03	7

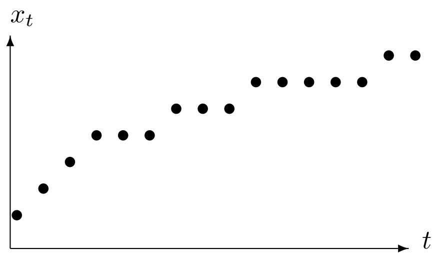
Analysis

We can summarise this by listing t and x_t together:

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x_t	1	2	3	4	4	4	5	5	5	6	6	6	6	6	7	7

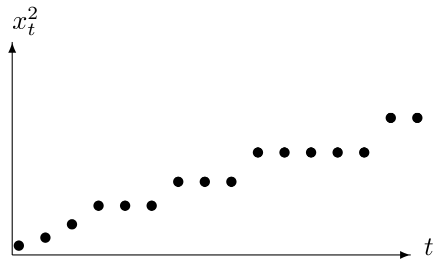
Does this data support the hypothesis that $t \propto x_t^2$?

We could plot the data as shown below — this seems to verify a quadratic relationship.



Alternatively, we could plot x_t^2 vs. t to see if we get a straight-line. This amounts to plotting the following:

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x_t	1	4	9	16	16	16	25	25	25	36	36	36	36	36	49	49



Finally, we could be more analytic, and note the following:

$$\begin{aligned} t &= x^2 \\ t' &= 2x \\ t'' &= 2 \\ t''' &= 0 \end{aligned}$$

So, we hope to take repeated differences in t, for fixed steps along x to estimate these values

$$\begin{array}{llll}
t & \Delta & \Delta^2 & \Delta^3 \\
t_0 & & & \\
 & t_0 - t_1 & & \\
t_1 & & t_0 - 2t_1 + t_2 & \\
 & t_1 - t_2 & & t_0 - 3t_1 + 3t_2 - t_3 \\
t_2 & & t_1 - 2t_2 + t_3 & \\
 & t_2 - t_3 & & t_1 - 3t_2 + 3t_3 - t_4 \\
t_3 & & t_2 - 2t_3 + t_4 & \\
 & t_3 - t_2 & & \\
t_4 & & &
\end{array}$$

we should see constant second-order differences (Δ^2), and zero third-order ones (Δ^3).

However we need values of t in terms of x, but re-organising our table gives the following:

x	t
1	0
2	1
3	2
4	3–5
5	6–8
6	9–13
7	14–15

We shall do the exercise three times, using the lowest value (of t for each x_t):

x_t	t	Δ	Δ^2	Δ^3
1	0			
		-1		
2	1		0	
		-1		0
3	2		0	
		-1		-2
4	3		2	
		-3		2
5	6		0	
		-3		-2
6	9		2	
		-5		
7	14			

the average value:

x_t	t	Δ	Δ^2	Δ^3
1	0			
		-1		
2	1		0	
		-1		-1
3	2		1	
		-2		0
4	4		1	
		-3		0
5	7		1	
		-4		1.5
6	11		-0.5	
		-3.5		
7	14.5			

and the largest value:

x_t	t	Δ	Δ^2	Δ^3
1	0			
		-1		
2	1		0	
		-1		-2
3	2		2	
		-3		2
4	5		0	
		-3		-2
5	8		2	
		-5		9
6	13		-7	
		2		
7	15			

We don't seem to exactly verify the law, but note that our data has lots of rounding, and these errors are large, and then get exacerbated by the repeated subtractions being done. The analysis using average values is not bad,however, if we ignore the extreme data values (x_1, x_7).

The problem is two-fold: the values we are using are too small and close together, and our x values are integral, so we give the same index when we find the pair of values | 0.51 | 0.01 | as we would if the values were | 1.2 | 0.49 |, yet in the first case the threshold value is toward the left-end of the leftmost x-slot, while in the second case it is towards the righthand side of the rightmost x-slot. If we used interpolation to get a better real-valued estimate in the slot for were the threshold was crossed, then we might get better fitting data.

The Key Questions

1. *What is the optimal value of k ?* For this task, k of about 0.5 is optimal. Values lower than this mean that we have to run many more time-step iterations in order to get adequate data.
2. *Are there "bad" values of k ?* Yes. Values of k above 0.5 lead to "nonsense" solutions in which the signal value appears to oscillate along the wire.
3. *Does the ℓ^2 law apply ?* It does, but this is hard to see, as has been just illustrated. It gets particularly hard to see if we use the 90% threshold value of 4.5V, from the Introduction to IC Design course (3BA4).

What's the problem with the 90% threshold ?

An issue that was not adequately covered in the setting of the problem, was the equation to use for the 20th x-slot. The equation in use was:

$$v_{n+1} = v_n + k \cdot (v_{n-1} - 2v_n + v_{n+1})$$

But what do we do for v_{20}, as there is value v_{21} ? Do we set $v_{21} = 0$? Or do we set $v_{21} = v_{20}$?

If we solve the problem using $v_{21} = 0$, this amounts to solving a different problem to the one set. With $v_{21} = 0$ we are looking at a wire, initially at 0v, with 5v suddenly applied to the left end, *and 0v applied to the right end* !

Initially this behaves like our wire with a free right end, but n the long run we get a different steady-state solution, with the wire voltage varying linearly form 5v at the left end to 0V at the right end. This is why the value of v_{20} never gets above $0.25V$, no matter how long the simulation runs.

The equation $v(x, t) = 5 - 0.25x$ is a solution to the diffusion equation, as

$$\frac{\partial(5 - 0.25x)}{\partial t} = 0 = \frac{\partial^2(5 - 0.25x)}{\partial x^2}$$

What should have been the equation for v_{20} ?

If we go back to the original derivation in 3BA4, then the last element has a resistor ($r\Delta x$) connecting x_{19} to x_{20} and a capacitor ($c\Delta x$) connecting x_{20} to ground, with no resistor leading out from x_{20} to x_{21} (because it doesn't exist !). The equation that we derive is therefore:

$$v_{20} = v_{19} + k \cdot (v_{19} - v_{20})$$

which is equivalent to the normal equation if we set $v_{21} = v_{20}$.

We need to be careful about boundaries and ends of the problems we try to solve.

We need to remember that a single differential equation may have many (quite) different solutions.

Relationship between Δx, Δt and k

Remember that

$$k = \frac{\Delta t}{rc\Delta x^2}$$

where Δ is the time-step, and Δx is the distance-step.

If our experiment was simulating the following thin wire:

$$\begin{aligned} \ell &= 2000\mu m \\ r &= 20\Omega/\mu m = 2 \cdot 10^7 \Omega/m \\ c &= 1fF/\mu m = 10^{-9}F/m \end{aligned}$$

Then we determine our steps as follows:

1. The distance-step Δx is simply the line length divided by the number of segments (20)

$$\Delta x = \ell/20 = \frac{2000}{20} = 100\mu m = 10^{-4}m$$

2. We then re-arrange the equation to give Δt in terms of the other values:

$$\Delta t = k \cdot rc\Delta x^2$$

3. We then use thus equation to get the time-step

$$\begin{aligned} \Delta t &= k \cdot rc\Delta x^2 \\ &= 0.5 \times 2 \cdot 10^7 \times 10^{-9} \times \left(10^{-4}\right)^2 \\ &= 10^7 \times 10^{-9} \times 10^{-8} \\ &= 10^{-10}s \\ &= 0.1nS \end{aligned}$$

1.15 3BA1-II Lecture 14

1.15.1 Solving Differential Equations Numerically

We shall consider 1st order differential equations (D.E.s).

Problem: we want to find y as a function of x ($y(x)$) given that we know that

$$y' = f(x,y)$$

Here f is a function describing the differential equation — it is not the solution, or its derivative. Alternative ways of writing $y' = f(x,y)$ are:

$$\begin{aligned} y'(x) &= f(x,y) \\ \frac{dy(x)}{dx} &= f(x,y) \end{aligned}$$

Working Example

We shall take the following D.E. as an example:

$$f(x,y) = y$$

or

$$y' = y$$

Like most D.E.s, this has an infinite number of solutions:

$$y(x) = C \cdot e^x \qquad \forall C \in \mathbb{R}$$

We can single out one solution by supplying an *initial condition*

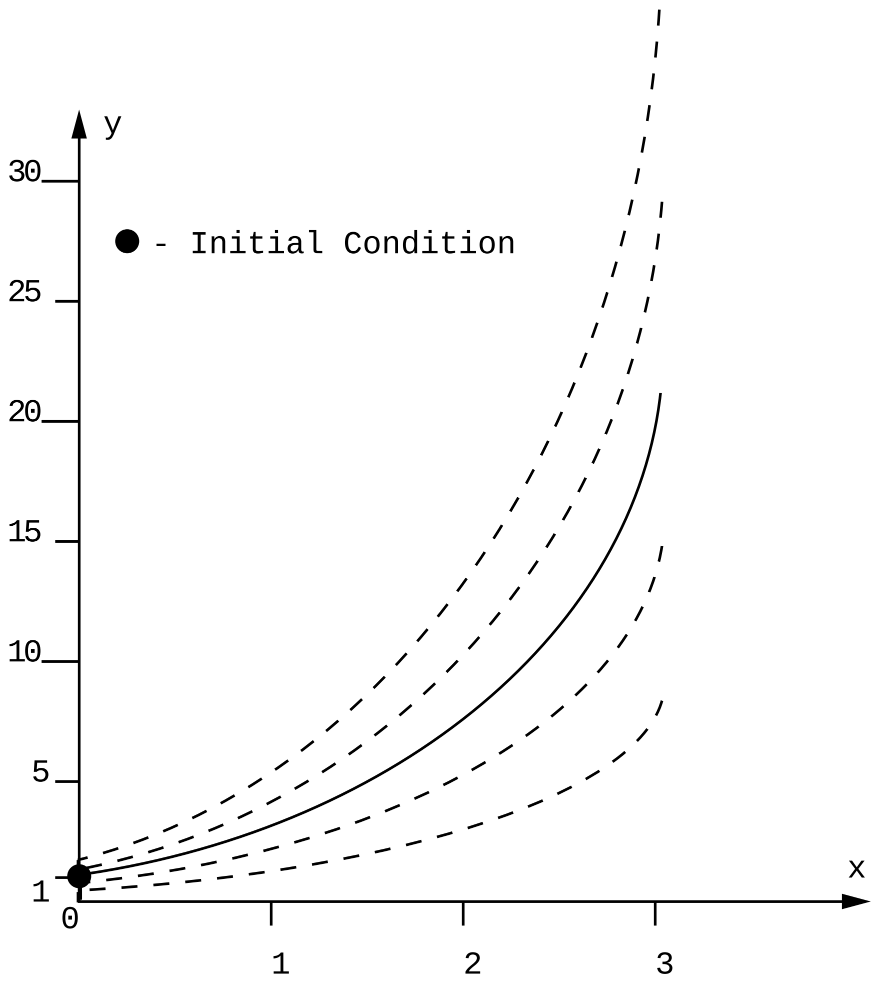
$$y(a) = \alpha$$

So, in our example, if we say that $y(0) = 1$, then we find that $C = 1$ and out solution is $y = e^x$.

We can now define the **Initial Condition Problem** as finding the solution $y(x)$ of

$$y' = f(x,y) \qquad y(a) = \alpha$$

The following graph shows some of the many solutions, and how the initial condition singles one out:



The dashed lines show the many solutions for different values of C. The solid line shows the solution singled out by the initial condition that $y(0) = 1$.

1.15.2 The Lipschitz Condition

We can give a condition that determines when the initial condition is sufficient to ensure a unique solution, known as the *Lipschitz Condition*:

For $a \leq x \leq b$, for all $-\infty < y, y^* < \infty$,

If there is an L such that

$$|f(x,y) - f(x,y^*)| \leq L\,|y - y^*|$$

Then the solution to $y' = f(x,y)$ is unique, given an initial condition.

L is often referred to as the *Lipschitz Constant*.

A useful estimate for L is to take $\left|\frac{\partial f}{\partial y}\right| \leq L$, for x in (a, b).

Example: given our example of $y' = y = f(x, y)$, then we can see do we get a suitable L.

$$\begin{aligned} \frac{\partial f}{\partial y} &= \frac{\partial(y)}{\partial(y)} \\ &= 1 \end{aligned}$$

So we shall try $L = 1$

$$\begin{aligned} |f(x, y) - f(x, y^*)| &= |y - y^*| \\ &\leq 1 \cdot |y - y^*| \end{aligned}$$

So we see that we satisfy the Lipschitz Condition with a Constant $L = 1$.

1.15.3 Numerically solving $y' = f(x, y)$

We assume we are trying to find values of y for x ranging over the interval $[a, b]$.

We shall solve this iteratively, starting with the one point were we have the exact answer, namely the initial condition:

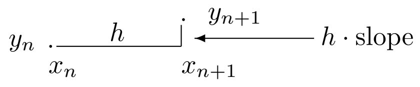
$$x_0 = a \qquad y_0 = y(x_0) = y(a) = \alpha$$

We shall generate a series of x-points from a to b, separated by a small step-interval h:

$$\begin{aligned} x_0 &= a \\ x_i &= a + ih \\ h &= \frac{b - a}{N} \\ x_N &= b \end{aligned}$$

So y_i will be the approximation to $y(x_i)$, the true value.

The technique works by using applying f at the current point (x_n, y_n) to get an estimate of y' at that point.



This is then used to compute y_{n+1} as follows:

$$y_{n+1} = y_n + h \cdot f(x_n, y_n)$$

This technique for solving D.E.'s is known as *Euler's Method.*

It is simple, slow and inaccurate, with experimentation showing that the error is $O(h)$.

In our example, we have

$$y' = y \qquad f(x,y) = y \qquad y_{n+1} = y_n + h \cdot y_n$$

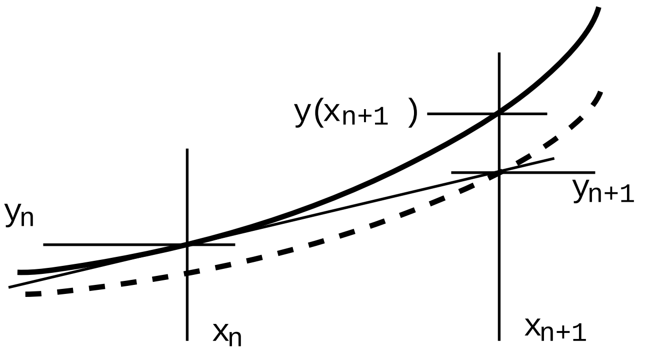
At each point after x_0, we accumulate an error, because we are using the slope at x_n to estimate y_{n+1}, which assumes that the slope doesn't change over interval $[x_n, x_{n+1}]$.

Truncation Errors

The error introduced at each step is called the *Local Truncation Error.*

The error introduced at any given point, as a result of accumulating all the local truncation errors up to that point, is called the *Global Truncation Error.*

We observe that the effect of each local truncation error is move our calculation from the correct solution, to one that is close by:



In the diagram above, the local truncation error is $y(x_{n+1}) - y_{n+1}$

We can estimate the local truncation error, by assuming the value y_n for x_n is exact as follows: as follows:

$$\begin{array}{ll} & y(x_{n+1}) \\ = & \quad \text{“ } x_{n+1} = x_n + h \text{ ”} \\ & y(x_n + h) \\ = & \quad \text{“ Taylor expansion about } x = x_n \text{ ”} \\ & y(x_n) + hy'(x_n) + \dfrac{h^2}{2} y''(\xi) \\ = & \quad \text{“ Assuming } y_n \text{ is exact } (y_n = y(x_n)), \text{ so } y'(x_n) = f(x_n, y_n) \text{ ”} \\ & y(x_n) + hf(x_n, y_n) + \dfrac{h^2}{2} y''(\xi) \end{array}$$

We then look at y_{n+1}:

$$\begin{array}{cl} & y_{n+1} \\ = & \quad \text{`` Euler's Method ''} \\ & y_n + hf(x_n, y_n) \end{array}$$

We subtract the two results above to get

$$y(x_{n+1}) - y_{n+1} = -\frac{h^2}{2}y''(\xi) = O(h^2)$$

as y'' is independent of h.

So we see that the local truncation error for Euler's Method is $O(h^2)$.

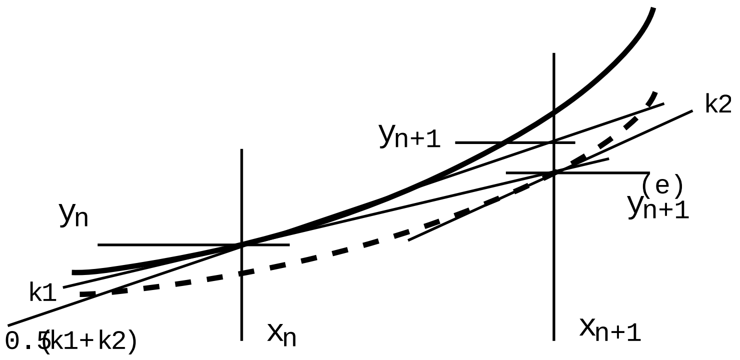
So halving h reduces the local error by a quarter, but we now require twice as many steps to get to a particular x-value, so the net effect is that the global error is only halved, and hence is $O(h)$.

As a general principle, we find that if the Local Truncation Error is $O(h^{p+1})$, then the Global Truncation Error is $O(h^p)$.

1.16 3BA1-II Lecture 15

1.16.1 Improved Differentiation Techniques

We can improve on Euler's technique to get better estimates for y_{n+1}. The idea is to use the equation $y' = f(x,y)$ to estimate the slope at x_{n+1} as well, and then average these two slopes to get a better result.



We let $y^{(e)}_{n+1}$ denote the value computed using Euler's Method. Here we see that $y'(x_n, y_n) = f(x_n, y_n)$ is the slope at x_n, while $y'(x_{n+1}, y^{(e)}_{n+1}) = f(x_{n+1}, y^{(e)}_{n+1})$ is the slope at x_{n+1}.

We use k_1 and k_2 to denote the changes in y due to multiplying the slopes by h in each case.

We define the quantities as follows:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ y^{(e)}_{n+1} &= y_n + k_1 \\ k_2 &= hf(x_{n+1}, y^{(e)}_{n+1}) \\ &= hf(x_n + h, y_n + k_1) \end{aligned}$$

We then compute y_{n+1} as

$$y_{n+1} = \frac{1}{2}(k_1 + k_2)$$

This technique is known as *Heun's Method.*

It can be shown to have a global truncation error that is $O(h^2)$.

The cost of this improvement in error behaviour is that we evaluate f twice on each h-step.

Runge-Kutta Techniques

We could repeat this exercise, trying to evaluate k_3 by applying f to x_{n+1} and the best value of y_{n+1} once more. We could do more, getting k_4, k_5, and so on.

This leads to a large class of improved differentiation techniques which evalaute f many times at each h-step, in order to get better error performance.

This class of techniques is referred to collectively as *Runge-Kutta* techniques, of which Heun's Method is the simplest example.

The classical Runge-Kutta technique evaluates f four times at each x_i, to get a method with global truncation error of $O(h^4)$.

1.16.2 Interpolation

Consider that we have been given $n+1$ data points:

$$(x_1, f_1), (x_2, f_2), \ldots (x_n, f_n), (x_{n+1}, f_{n+1})$$

and we want to find a function P such that

$$P(x_i) = f_i \qquad \forall i \in 1..n+1$$

We hope that $P(x)$ will be a good approximation to some underlying (unknown) function f that generated or describes the data.

If we use P to evaluate $f(x)$ inside the interval formed by $x_1..x_{n+1}$, then we are doing *Interpolation.* If we evaluate for x outside this interval, then we are performing *Extrapolation.*

Polynomial Interpretation

Usually, P is a polynomial, of degree n

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

where n is chosen suitably.

Exploring a few examples ($n+1 = 2$, $n+1 = 3$,..) soon makes it clear that in general we a polynomial of degree n to fit data with $n+1$ points. It also becomes clear that with polynomials of higher degree than this, that we have infinitely many which fit the data supplied, which is not helpful, as we do not then know which polynomial best matches f for values of x not in the data-set.

We exploit the following theorem:

Given $n+1$ data-points, there is a unique polynomial of degree $\leq n$ that passes through these points.

It is clear that this unique polynomial is the most useful on to find.

How do we determine the coefficients of this polynomial.

We could take the polynomial as

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

and plug in our data-values to get

$$f_i = a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n \qquad \forall i \in 1..n+1$$

which would give us $n+1$ equations in $n+1$ unknowns $(a_0, .., a_n)$ which we could then solve in the usual fashion.

An easier approach is best illustrated by considering the cases of $n+1 = 2$ and $n+1 = 3$:

Case $n+1 = 2$**.** P will be of degree 1, and so we write P in the following form:

$$P(x) = C_0 + C_1(x - x_1)$$

We then insert $x = x_1$ to get:

$$f_1 = P(x_1) = C_0 + C_1(x_1 - x_1) = C_0$$

We can then substitute for C_0 in P and then evaluate at x_2 to get:

$$f_2 = P(x_2) = f_1 + C_1(x_2 - x_1)$$

which can be manipulated to solve for C_1 as follows:

$$C_1 = \frac{f_2 - f_1}{x_2 - x_1}$$

Case $n+1 = 2$**.** P will be of degree 2, and so we write P in the following form:

$$P(x) = C_0 + C_1(x - x_1) + C_2(x - x_1)(x - x_2)$$

We then insert $x = x_1$ to get

$$f_1 = P(x_1) = C_0 + C_1(x_1 - x_1) + C_2(x_1 - x_1)(x_1 - x_2) = C_0$$

We can then substitute for C_0 in P and then evaluate at x_2 to get:

$$\begin{aligned} f_2 &= P(x_2) \\ &= f_1 + C_1(x_2 - x_1) + C_2(x_2 - x_1)(x_2 - x_2) \\ &= = f_1 + C_1(x_2 - x_1) \end{aligned}$$

which can be manipulated to solve for C_1 as follows

$$C_1 = \frac{f_2 - f_1}{x_2 - x_1}$$

Finally we insert $x = x_3$ to get

$$\begin{aligned} f_3 &= P(x_3) \\ &= C_0 + C_1(x_3 - x_1) + C_2(x_3 - x_1)(x_3 - x_2) \\ &= f_1 + \frac{f_2 - f_1}{x_2 - x_1}(x_3 - x_1) + C_2(x_3 - x_1)(x_3 - x_2) \end{aligned}$$

This can be manipulated to solve for C_2 as follows:

$$f_3 - f_1 - \frac{f_2 - f_1}{x_2 - x_1}(x_3 - x_1)$$

We notice two things — the derivation of C_0 and C_1 is the same regardless of n, and the equations get quite complex.

In general, to get an interpolating polynomial of degree n we start with the form:

$$P(x) = C_0 + \sum_{i=1}^{n} C_i(x - x_1)(x - x_2)\cdots(x - x_n)$$

and use $x = x_i$ to solve for the C_i as described above.

Using Talyor's and Rolle's Theorem, it is possible to show that the truncation error R_T is

$$\begin{aligned} R_T &= f(x) - P(x) \\ &= \frac{f^{(n+1)}(\xi)}{n+1!}(x - x_1)(x - x_2)\cdots(x - x_n) \end{aligned}$$

where ξ is somewhere in the interval $x_1..x_{n+1}$.

Note that it is a consequence of this error expression that $R_T = 0$ for $x = x_i$, for $i = 1..n + 1$.

Interpolating known functions

Sometime we use interpolation to get a polynomial approximation to known functions.

Why? Often the polynomial approximation is much easier to compute !

Given f, are we better of using polynomials of higher degree, in order to get better accuracy ?

The answer surprisingly is No. Runge found that the function

$$f(x) = \frac{1}{1 + 25x^2}$$

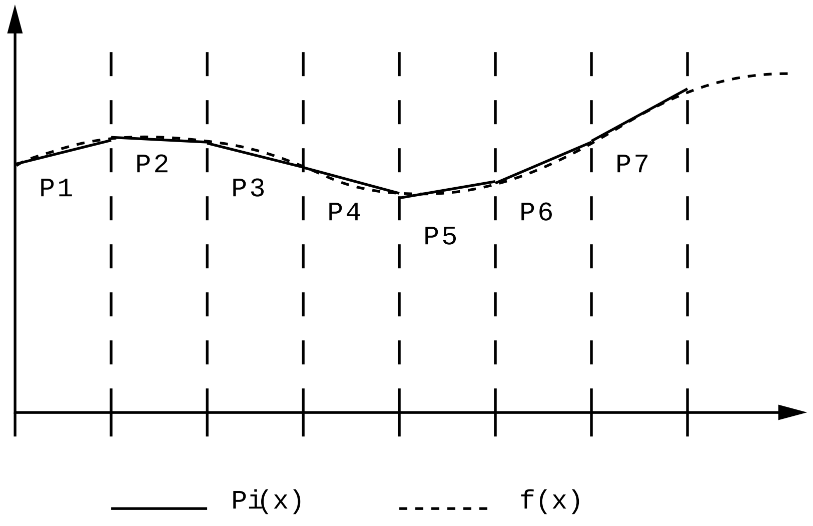
on the interval $[-1, +1]$ has the surpirising property, that as the degree of the interpolating polynomial rises, so does the error at points in between the interpolation points. In fact he found that the error tends to infinity as $n \to \infty$.

An even stronger result shows that for any interpolating polynomial P, there is a function ψ such that $\psi(x) = P(x)$ at the interpolation points, but that the difference $\psi(x) - P(x)$ gets arbitrarily large at some in-between points.

The consequence of this is that we should **only use P of low degree for interpolation**.

In practise we $n = 1$ (linear interpolation) or $n = 2$ (quadratic interpolation) and us this to repeatedly approximate f over short sections.

The following diagram shows a function $f(x)$ being interpolated in a *piecewise linear* fashion by straight-line segments $P1(x)$, $P2(x)$, $\ldots Pn(x)$, each of degree 1.



1.17 3BA1-II Lecture 16

1.17.1 Numerical Integration

Remember integrating f means finding F such that:

$$\int_a^b f(x)dx = F(x) \mid_a^b = F(b) - F(a)$$

Two possible reasons for integrating numerically are that

- the function F is expensive to compute
- the function F has no analytic form (e.g. $f = e^{-x^2}$).

In practise, we divide the interval $b - a$ by n to get slots of width h

$$h = \frac{b-a}{n} \qquad x_i = a + ih$$

We let f_i denote $f(x_i)$, and set $x_0 = a$ and $x_n = b$.

In general we evaluate integrals piecewise, by taking k slots at a time ($k+1$ data points) and using an interpolating polynomial P_k of degree k to approximate f over that interval. We can then integrate the polynomial to get I_k, and evaluate this at the end-points to get an approximation to the integral:

$$\begin{aligned}
P_k &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\
I_k &= a_0 x + \frac{a_1 x^2}{2} + \frac{a_2 x^3}{3} + \cdots + \frac{a_n x^{n+1}}{n+1} \\
\int_a^b P_k(x)dx &= I_k(b) - I_k(a)
\end{aligned}$$

Having done this for $x_0..x_k$, we then repeat for $x_k..x_{2k}$, and then for $x_{2k}..x_{3k}$, an so on, until we reach x_n.

Best Choice of k for Integration

As with polynomial interpolation, we find that polynomials of high degree do not give better accuracy. Indeed, for Runge's formula:

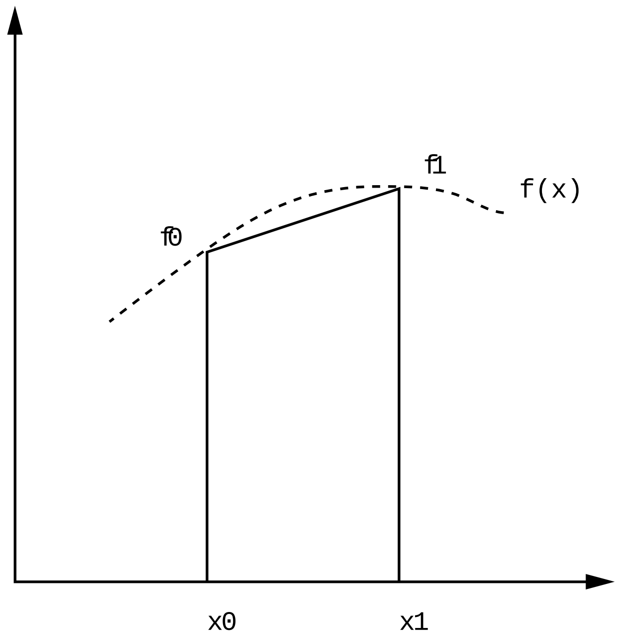
$$f(x) = \frac{1}{1 + 25x^2}$$

we find that the error when integrating increases as k gets larger.

In practise we restrict k to 1 or 2.

1.17.2 Trapezoidal Rule ($k = 1$)

We have one slot, with two data points, and we approximate the integral by the trapezoidal shape formed by those two points:



The area under the straight line is the area of the box of height f_0 plus that if the triangle on top of height $x_1 - x_0$, both of width h:

$$\begin{aligned} \text{area} &= h \cdot f_0 + \frac{1}{2}h \cdot (f_1 - f_0) \\ &= hf_0 + \frac{1}{2}hf_1 - \frac{1}{2}hf_0 \\ &= \frac{h}{2}(f_0 + f_1) \end{aligned}$$

So the trapezoidal rule states that

$$\int_{x_0}^{x_1} f(x)dx = \frac{h}{2}(f_1 + f_0) + R_T$$

where R_T is the truncation error.

We can adapt the law from the previous lecture that gave R_T for P_k to get a law giving R_T for I_k:

$$R_T = \int_{x_0}^{x_k} (x - x_0)(x - x_1) \cdots (x - x_k) \frac{f^{(k+1)}(\xi)}{n + 1!} dx$$

For the trapezoidal rule ($k = 1$) this gives us:

$$R_T = \int_{x_0}^{x_1} (x - x_0)(x - x_1)\frac{f''(\xi)}{2!}dx$$

We solve this as follows:

$$\begin{array}{ll}
 & \displaystyle\int_{x_0}^{x_1} (x - x_0)(x - x_1)\frac{f''(\xi)}{2!}dx \\
= & \text{“ change of variable } x = x_0 + ph\text{, nothing that } x_1 = x_0 + h \text{ ”} \\
 & \displaystyle h\int_0^1 ph(p-1)h\frac{f''(\xi)}{2}dp \\
= & \text{“ Mean Value Theorem, Integral Version ”} \\
 & \displaystyle \frac{h^3 f''(\eta)}{2}\int_0^1 p(p-1)dp \\
= & \text{“ flatten integrand ”} \\
 & \displaystyle \frac{h^3 f''(\eta)}{2}\int_0^1 (p^2 - p)dp \\
= & \text{“ integrate ”} \\
 & \displaystyle \frac{h^3 f''(\eta)}{2}\left(\frac{p^3}{3} - \frac{p^2}{2}\right)\Big|_0^1 \\
= & \text{“ evaluate ”} \\
 & \displaystyle \frac{h^3 f''(\eta)}{2}\left(\frac{1}{3} - \frac{1}{2}\right) \\
= & \text{“ simplify ”} \\
 & \displaystyle \frac{h^3 f''(\eta)}{12}
\end{array}$$

So we see that the truncation error per trapezoidal step is $O(h^3)$.
We can apply the trapezoidal rule over the entire range $x_0..x_n$ in one go:

$$\int_{x_0}^{x_n} f(x)dx = h(\frac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{n-1} + \frac{1}{2}f_n)$$

In fact given $n + 1$ data points, it is much easier to integrate the function they represent than to interpolate to find that function !

1.17.3 Simpson's Rule ($k = 2$)

Simpson's rule uses a parabola (polynomial of degree 2) to fit three points:
The resulting formula obtained is:

$$\int_{x_0}^{x_2} f(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) + R_T$$

1.17.4 Romberg's Method

Romberg's Method is based on combining the above methods with Richardson Extrapolation to improve the error bounds. It has the nice property that the calculation rounding error R_{XF} for the integral

$$\int_a^b f(x)dx$$

is bounded by $(b-a)\epsilon$:

$$|R_{XF}| \leq |b-a|\,\epsilon$$

where ϵ is the relative rounding error of the arithmetic system in use. This error bound is about as good as it is reasonably possible to expect.

We do not cover Romberg's methods here.

Appendix A

Calculus Prerequisites

Taken from Mathews & Fink, *for information purposes only.*

A.1 Limits and continuity

$\lim_{x \to x_0} f(x) = L$

$\quad \widehat{=} \quad \forall \epsilon > 0 \bullet \exists \delta > 0 \bullet 0 < \mid x - x_0 \mid < \delta \Rightarrow \mid f(x) - L \mid < \epsilon$

$\lim_{h \to 0} f(x_0 + h) = L$

$\quad \equiv \quad \lim_{x \to x_0} f(x) = L$

f **continuous at** x_0

$\quad \widehat{=} \quad \lim_{x \to x_0} f(x) = f(x_0)$

f **continuous on** S

$\quad \widehat{=} \quad \forall x \in S \bullet f$ **continuous at** x

$[a, b]$

$\quad \widehat{=} \quad \{\, x \mid a \leq x \leq b \,\}$

$C^0(S)$

$\quad \widehat{=} \quad \{\, f \mid f \in \mathbb{R} \to \mathbb{R} \wedge f$ **continuous on** $S \,\}$

$C^n(S)$

$\quad \widehat{=} \quad \{\, f \mid f \in C^{n-1}(S) \wedge f^{(n)}$ **continuous on** $S \,\}$

$C(S)$

$\quad \widehat{=} \quad C^1(S)$

Let $\{ x_n \}_{n=1}^{\infty}$ be an infinite sequence:

$$\begin{aligned}
&\lim_{n\to\infty} x_n = L \\
&\quad \widehat{=} \quad \forall \epsilon > 0 \bullet \exists N \bullet n > N \Rightarrow \mid x_n - L \mid < \epsilon \\
&x_n \to L \textbf{ as } n \to \infty \\
&\quad \widehat{=} \quad \lim_{n\to\infty} (x_n - L) = 0 \\
&\{ \epsilon_n \}_{n=1}^{\infty} \\
&\quad \widehat{=} \quad \{ x_n - L \}_{n=1}^{\infty}
\end{aligned}$$

$$\begin{gathered}
f \textbf{ defined on } S \wedge x_0 \in S \\
\Rightarrow \\
\left(\begin{gathered} f \textbf{ continuous at } x_0 \\ \equiv \\ \lim_{n\to\infty} x_n = x_0 \Rightarrow \lim_{n\to\infty} f(x_n) = f(x_0) \end{gathered} \right)
\end{gathered}$$

A.1.1 Intermediate Value Theorem

$$\begin{gathered}
f \in C[a,b] \\
\Rightarrow \\
\forall L \textbf{ between } f(a) \textbf{ and } f(b) \bullet \exists c \in (a,b) \bullet f(c) = L
\end{gathered}$$

A.1.2 Extreme Value Theorem

$$\begin{gathered}
f \in C[a,b] \\
\Rightarrow \\
\exists M_1, M_2 \bullet \exists x_1, x_2 \in [a,b] \bullet \forall x \in [a,b] \bullet M_1 = f(x_1) \le f(x) \le f(x_2) = M_2
\end{gathered}$$

$$\begin{aligned}
M_1 = \quad f(x_1) \quad &= \min_{a \le x \le b} \{ f(x) \} \\
M_2 = \quad f(x_2) \quad &= \max_{a \le x \le b} \{ f(x) \}
\end{aligned}$$

A.2 Differentiable Functions

Assume f defined on open interval containing x_0:

$$\begin{aligned}
&\textbf{derivative of } f \textbf{ at } x_0 \\
&\quad \widehat{=} \quad \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}
\end{aligned}$$

$$\begin{gathered} f \textbf{ differentiable at } x_0 \\ \Rightarrow \\ f \textbf{ continuous at } x_0 \end{gathered}$$

A.2.1 Rolle's Theorem

$$\begin{gathered} f \in C[a,b] \ \wedge \ f' \textbf{ defined over } \text{(a,b)} \wedge f(a) = f(b) = 0 \\ \Rightarrow \\ \exists c \in (a,b) \bullet f'(c) = 0 \end{gathered}$$

A.2.2 Mean Value Theorem

$$\begin{gathered} f \in C[a,b] \ \wedge \ f' \textbf{ defined over } \text{(a,b)} \\ \Rightarrow \\ \exists c \in (a,b) \bullet f'(c) = \tfrac{f(b)-f(a)}{b-a} \end{gathered}$$

Integral form of M.V.T

$$\begin{gathered} f \in C[a,b] \ \wedge \ g \in C[a,b] \ \wedge \ \mathsf{sign}(g) \text{ unchanged in } [a,b] \\ \Rightarrow \\ \exists c \in [a,b] \bullet \textstyle\int_a^b f(x)g(x)dx = f(c) \int_a^b g(x)dx \end{gathered}$$

A.2.3 Generalised Rolle's Theorem

$$\begin{gathered} f \in C[a,b] \\ \wedge \\ f', f'', \ldots, f^{(n)} \textbf{ defined over } \text{(a,b)} \\ \wedge \\ (\exists x_0, x_1, \ldots, x_n \in [a,b] \bullet f(x_j) = 0, \ \textbf{for } j = 0, 1, \ldots n) \\ \Rightarrow \\ \exists c \in (a,b) \bullet f^{(n)}(c) = 0 \end{gathered}$$

A.3 Integrals

A.3.1 First Fundamental Theorem

$$\begin{gathered} f \textbf{ continuous on } [a,b] \wedge F'(x) = f(x), x \in [a,b] \\ \Rightarrow \\ \textstyle\int_a^b f(x)dx = F(b) - F(a) \end{gathered}$$

A.3.2 Second Fundamental Theorem

$$f \textbf{ continuous on } [a,b] \wedge x \in (a,b)$$
$$\Rightarrow$$
$$\tfrac{d}{dx}\int_a^x f(t)dt = f(x)$$

A.3.3 Mean Value Theorem for Integrals

$$f \in C[a,b]$$
$$\Rightarrow$$
$$\exists c \in (a,b) \bullet \tfrac{1}{b-a}\int_a^b f(x)dx = f(c)$$

A.3.4 Weighted Interval Mean Value Theorem

$$f, g \in C[a,b] \wedge g(x) > 0, x \in [a,b]$$
$$\Rightarrow$$
$$\exists c \in (a,b) \bullet \int_a^b f(x)g(x)dx = f(c)\int_a^b g(x)dx$$

A.4 Series

Let $\{ a_n \}_{n=1}^{\infty}$ be a sequence. Then $\sum_{n=1}^{\infty} a_n$is a **series**. The nth partial sum is $S_n = \sum_{k=1}^{n} a_k$

$$\lim_{n\to\infty} S_n$$
$$\widehat{=} \quad \lim_{n\to\infty}\sum_{k=1}^{n} a_k = S$$

A.4.1 Taylor's Theorem

$$f \in C^{n+1}[a,b] \wedge x_0 \in [a,b]$$
$$\Rightarrow$$
$$\forall x \in (a,b) \bullet \exists c \textbf{ between } x \textbf{ and } x_0 \bullet f(x) = P_n(x) + R_n(x)$$

where

$$P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k$$

and

$$R_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x-x_0)^{n+1}$$

Note that

$$P_n^{(k)}(x_0) = f^{(k)}(x_0), \text{ for } k = 0, 1, \ldots, n$$

A.5 Evaluation of a Polynomial

Let

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x^1 + a_0 x^0$$

A.5.1 Horner's Method for Polynomial Evaluation

Then

$$P(c) = (((\ldots((a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_2)x + a_1)x + a_0$$

```
r := a[n];
for(k=n-1, k--, k==0)
  r := r*x + a[k]
```

A.6 Taylor's Theorem revisited

$$\begin{gathered} f \in C^{n+1}[a,b] \wedge x_0 \in [a,b] \\ \Rightarrow \\ \forall x \in (a,b) \bullet \exists c \textbf{ between } x \textbf{ and } x_0 \bullet f(x) = P_n(x) + R_n(x) \end{gathered}$$

where

$$P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

and

$$R_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x - x_0)^{n+1}$$

Note that

$$P_n^{(k)}(x_0) = f^{(k)}(x_0), \text{ for } k = 0, 1, \ldots, n$$

A.7 Laws of Differentiation

$f(x)$	$\dfrac{df(x)}{dx}$
x^n	$n \cdot x^{n-1}$
e^x	e^x
$\ln(x)$	$\dfrac{1}{x}$
$\sin(x)$	$\cos x$
$\cos(x)$	$-\sin(x)$

$$\frac{df(x) \pm g(x)}{dx} = \frac{df(x)}{dx} \pm \frac{dg(x)}{dx}$$

or

$$d(u+v) = du + dv$$

$$\frac{df(x) \cdot g(x)}{dx} = f(x)\frac{dg(x)}{dx} + g(x)\frac{df(x)}{dx}$$

or

$$d(u \cdot v) = u \cdot dv + v \cdot du$$

$$\frac{d\left(\frac{f(x)}{g(x)}\right)}{dx} = \frac{g(x)\frac{df(x)}{dx} - f(x)\frac{dg(x)}{dx}}{g(x)^2}$$

or

$$d\left(\frac{u}{v}\right) = \frac{v \cdot du - u \cdot dv}{v^2}$$

$$\frac{df(g(x))}{dx} = \frac{f(g(x))}{dfg(x)} \cdot \frac{dg(x)}{dx} = \frac{f(y)}{dy} \cdot \frac{dg(x)}{dx} \ \textbf{where}\ y = g(x)$$

Appendix B

Haskell Experimentation for 3BA1 Part II

This appendix gives both the sources and outcomes of a number of experiments using the programming language Haskell (`www.haskell.org`) to perform numerical algorithms.

The code, comments and results are presented here for information purposes only.

```
> module Experiments
> where
```

B.1 Representation Errors

B.1.1 Associativity

$$a + (b + c) \neq (a + b) + c$$

```
> assoc_error
>  = ((x,y),(y-x,x==y))
>  where
>   x = a+(b+c)
>   y = (a+b)+c
>   a = 1.0
>   b = 3.0E-7
>   c = 3.0E-7
```

Hugs Nov 2002, Pentium III, WinXp:

```
Experiments> assoc_error
((1.0,1.0),(1.19209e-007,False))
```

B.1.2 Repeated Summing

$$\sum_{i=1}^{100,000} 0.1$$

```
> ksum n k
>  = ksum' n k 0
>  where
>    ksum' n k s
>     | n <= 0    = s
>     | otherwise = ksum' (n-1) k $! (k+s)
```

Hugs Nov 2002, Pentium III, WinXp:

```
Experiments> ksum 100000 0.1
9998.56
```

B.1.3 Root Finding

We solve $x - e^{-x} = 0$, using initial guess of $x^* \in [0.5, 0.6]$ using different techniques. We need both f and f':

```
> f :: Double -> Double
> f x = x - exp (-x)

> f' :: Double -> Double
> f' x = 1 + exp(-x)
```

For the interval techniques we provide an interval builder that puts the bounds in order:

```
> mkIV (x1,x2)
>  | x1 <= x2   =  (x1,x2)
>  | otherwise  =  (x2,x1)
```

For the fixed point technique we need some equivalent fixed-point functions for f — namely φ_1, φ_2 and φ_3.

$$\begin{aligned}\varphi_1(x) &= e^{-x}\\ \varphi_2(x) &= -log x\\ \varphi_3(x) &= x - f(x)/f'(x)\end{aligned}$$

```
> phi1 x = exp (-x)
> phi2 x = -(log x)
> phi3 x = x - (f x)/(f' x)
```

Bisection Method

Finding the midpoint $x_{n+1} = (x_{n-1} + x_n)/2$:

```
> mid :: (Double,Double) -> Double
> mid (xl,xr) = (xl+xr)/2
```

Determining the sign of a value:

```
> sign :: Double -> Int
> sign x
>  | x <  0.0  =  -1
>  | x == 0.0  =  0
>  | x >  0.0  =  1
```

A single step of the bisection method:

```
> bisectStep f (xl,xr)
>  | sign (f xm) /= sign (f xl)  =  (xl,xm)
>  | otherwise                   =  (xm,xr)
>  where
>    xm = mid (xl,xr)
```

We don't need to use `mkIV` here because our algorithm keeps the bounds ordered, as long as they are on entry to the function. The iterative solver ensures this is the case before beginning:

```
> bisect f iv = iterate (bisectStep f) (mkIV iv)
```

Outcome:

```
Experiments> take 18 (bisect f (0.5,0.6))
[(0.5,0.6),(0.55,0.6),(0.55,0.575),(0.5625,0.575),(0.5625,0.56875)
,(0.565625,0.56875),(0.565625,0.567188),(0.566406,0.567188)
,(0.566797,0.567188),(0.566992,0.567188),(0.56709,0.567188)
,(0.567139,0.567188),(0.567139,0.567163),(0.567139,0.567151)
,(0.567139,0.567145),(0.567142,0.567145),(0.567142,0.567143)
,(0.567143,0.567143)]
```

Regula Falsi

First, the secant computation:

```
> sec f (xl,xr)
>  = xr - fr*(xr-xl)/(fr-fl)
>  where
>    fl = f xl
>    fr = f xr
```

Next, the Regula-Falsi step:

```
> regFalsiStep f (xl,xr)
>  | sign (f xm) /= sign (f xl)  =  (xl,xm)
>  | otherwise                   =  (xm,xr)
>  where
>    xm = sec f (xl,xr)
```

Finally, the complete method:

```
> regulaFalsi f iv = iterate (regFalsiStep f) (mkIV iv)
```

The outcome:

```
Experiments> take 7 (regulaFalsi f (0.5,0.6))
[(0.5,0.6),(0.5,0.567545),(0.5,0.567148),(0.5,0.567143)
,(0.5,0.567143),(0.567143,0.567143),(0.567143,0.567143)]
```

Secant Method

```
> secantStep f (xl,xr)
>  = mkIV(xr,xn)
>  where
>    xn = sec f (xl,xr)

> secant f iv = iterate (secantStep f) (mkIV iv)
```

Outcome:

```
Experiments> take 6 (secant f (0.5,0.6))
[(0.5,0.6),(0.567545,0.6),(0.567141,0.6),(0.567143,0.6)
,(0.567143,0.6),(0.567143,0.6)]
```

Newton-Raphson

```
> nrStep f f' x = x - (f x)/(f' x)

> newtonRaphson f f' x0 = iterate (nrStep f f') x0
```

Outcome:

```
Experiments> take 4 (newtonRaphson f f' 0.5)
[0.5,0.566311,0.567143,0.567143]
Experiments> take 4 (newtonRaphson f f' 0.6)
[0.6,0.56695,0.567143,0.567143]
```

Fixed-Point iteration

We simply use the Haskell built-in function `iterate` with our φ function and initial guess.
Outcomes for $\varphi_i, i \in 1, 2, 3$:

```
Experiments> take 24 (iterate phi1 0.5)
[0.5,0.606531,0.545239,0.579703,0.560065,0.571172,0.564863
,0.568438,0.566409,0.56756,0.566907,0.567277,0.567067
,0.567186,0.567119,0.567157,0.567135,0.567148,0.567141,0.567145
,0.567142,0.567144,0.567143,0.567143]

Experiments> iterate phi2 0.5
[0.5,0.693147,0.366513,1.00372,-0.00371457,
Program error: {primLogDouble (-0.00371457)}

Experiments> take 4 (iterate phi3 0.5)
[0.5,0.566311,0.567143,0.567143]
```

B.1.4 Difference Equations

Forward Difference

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

```
> dPlus f h x = (f(x+h) - f(x)) / h
```

Backward Difference

$$D_-(h) = \frac{f(x) - f(x-h)}{h}$$

```
> dMinus f h x = (f(x) - f(x-h)) / h
```

Central Difference

$$D_0(h) = \frac{f(x+h) - f(x-h)}{2h}$$

```
> d0 f h x = (f(x+h) - f(x-h)) / (2*h)
```

Shrinking h to improve $D_0(h)$

We take as example, $f(x) = e^x$, so that $f'(x) = e^x$, and we evaluate $f'(1) = e$.

$$D_0(h) = \frac{e^{1+h} - e^{1-h}}{2h}$$

```
> dExp0 x h  = (exp(x+h) - exp(x-h)) / (2*h)

> rExp0 x h  = (dExp0 x h) - exp x
```

Outcome:

```
Experiments> map (dExp0 1) [0.4,0.2,0.1,0.05]
[2.79135,2.73644,2.72282,2.71941]
Experiments> map (rExp0 1) [0.4,0.2,0.1,0.05]
[0.0730698,0.0181584,0.00453353,0.00113249]
```

We now consider looking at small values of h:

```
> smallhs = iterate (/10) 0.75
```

Outcome:

```
Experiments> take 9 (map (dExp0 1) smallhs)
[2.98038,2.72083,2.71831,2.71813,2.71797,2.71797,2.70208,3.17891,0.0]
Experiments> take 9 (map (rExp0 1) smallhs)
[0.262103,0.00255108,2.38419e-005,-0.000150919,-0.000310183
,-0.000309944,-0.0162046,0.460633,-2.71828]
```

Richardson Extrapolation

The first Richardson Extrapolation, for Central Differences is:

$$D_0(h) + \frac{D_0(h) - D_0(2h)}{3}$$

```
> extrRichardson  f h x = ( 4*(d0 f h x) - d0 f (2*h) x)/3

> richExp0 x h = extrRichardson  exp h x

> erichExp0 x h = (richExp0 x h) - exp x
```

Outcome:

```
Experiments> take 10 (map (richExp0 1) smallhs)
[2.68763,2.71828,2.71828,2.71808,2.71797,2.71532,2.67559,3.17891,0.0,0.0]
Experiments> take 10 (map (erichExp0 1) smallhs)
[-0.0306537,-2.38419e-007,-2.6226e-006,-0.000203848,-0.000310183
,-0.00295901,-0.0426958,0.460633,-2.71828,-2.71828]
```

We see that it gives more accurate answers than central difference for large-ish h but is prone to the same rounding problem.

We now turn our attention to the complete technique.

B.1.5 Full Richardson Extrapolation

Given $F(h)$, computable for $h \neq 0$ are trying to compute

$$\lim_{h \to 0} F(h)$$

We assume we have a way of calculating F which satisfies the following expansion in terms of h:

$$F(h) = F(0) + a_1 h^{p_1} + a_2 h^{p_2} + a_3 h^{p_3} + \cdots$$

where $a_1, a_2, \ldots$ are unknown, but $p_1, p_2, \ldots$ are known.

To find the limit, we supply the algorithm with F, the p_i an initial value of $h = h_0$, and a step reduction factor q, usually integral.

We define intermediate calculations as follows:

$$\begin{array}{rcl} F_1(H) & \widehat{=} & F(h) \\ F_{i+1}(h) & = & F_i(h) + \dfrac{1}{q^{p_i} - 1}(F_i(h) - F_i(qh)) \\ & = & F_i(h) + \dfrac{\Delta_i}{q^{p_i} - 1} \\ & & \textbf{where} \\ \Delta_i & = & (F_i(h) - F_i(qh)) \end{array}$$

We shall use the following extrapolation scheme:

F_1	$\Delta_1/(q^{p_1} - 1)$	F_2	$\Delta_2/(q^{p_2} - 1)$	F_3
$F_1(q^2h)$				
$F_1(qh)$		$F_2(qh)$		
$F_1(h)$		$F_2(h)$		$F_3(h)$

which we evaluate row-by row, stopping when adjacent values in the same *column* are within the desired (relative) error (ϵ).

We want a function that given ϵ, F, $\rho = \{1 \mapsto p_1, 2 \mapsto p_2, \ldots\}$, q and h_0, performs this extrapolation:

$$\begin{array}{rcl} \text{Richardson} & : & \mathbb{R} \to (\mathbb{R} \to \mathbb{R}) \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{R} \to \mathbb{R} \\ \text{Richardson } \epsilon\ F\ \rho\ q\ h_0 & \widehat{=} & x\ \textbf{where} \ldots \end{array}$$

```
> richardson r eps reps f rho q h0
>  = 0.0
>  where
>    hs = q*q*h0:q*h0:h0:(iterate (/q) h0)
>    fs     1 = map f hs
>    fs (i+1) = addzip (tail(fs i)) (dlt i)
>    dlt i = map (/den) (diff (fs i))
>            where den = (q ^ (rho i) - 1)

> diff (x1:rest@(x2:xs)) = (x2-x1):(diff rest)

> addzip (x:xs) (y:ys) = (x+y):(addzip xs ys)

> close reps x1 x2 = abs ((x1-x2)/(max x1 x2))  <= reps
```

B.1.6 Solving the Diffusion Equation

We want to solve the following equation for $v(x,t)$

$$rc\frac{\partial v}{\partial t} = \frac{\partial^2 v}{\partial x^2}$$

subject to the boundary conditions:

$$\begin{aligned} v(0,t) &= V_0 \\ v(x,0) &= 0, \quad 0 < x \leq \ell \end{aligned}$$

to determine the time t_r when $v(\ell, t_r) = 0.9V_0$.
We would then like to investigate how t_r varies with ℓ, for given r and c.

An initial investigation

The diffusion equation above was obtained from the following mixed differential/difference equation:

$$rc\frac{\partial v_i}{\partial t} = \frac{v_{i-1} - v_i}{\triangle x} - \frac{v_i - v_{i+1}}{\triangle x}$$

We can convert it into a difference equation by indexing over x and t to get:

$$rc\frac{v_{(i,t+1)} - v_{(i,t)}}{\triangle t} = \frac{v_{(i-1,t)} - v_{(i,t)}}{\triangle x} - \frac{v_{(i,t)} - v_{(i+1,t)}}{\triangle x}$$

We can then re-arrange this to give $v_{(i,t+1)}$ in terms of the values at time t:

$$v_{(i,t+1)} = v_{(i,t)} + \frac{\triangle t}{rc\triangle x}(v_{(i-1,t)} - 2 \cdot v_{(i,t)} + v_{(i+1,t)})$$

For the last section, we note that we need the formula:

$$v_{(n,t+1)} = v_{(n,t)} + \frac{\triangle t}{rc\triangle x}(v_{(n-1,t)} - v_{(n,t)})$$

This is easily implementable in a spreadsheet like Excel, and using conditional formatting it is easy to see that the propagation time down x is $O(\ell^2)$. An interesting result of that experimentation is that we get sensible answers provided that

$$\frac{\Delta t}{rc\Delta x} \leq 0.5$$

We can define a Haskell function to solve this, given the following inputs:

Input Voltage `v0`

Number of Segments `n`

Threshold `lvl` — this should be less that `v0`

Step Factor `k` — this is $\frac{\Delta t}{rc\Delta x}$

It outputs an infinite list of numbers, each indicating the largest index during the corresponding time-step of an segment value greater than the threshold. We initialise the starting segment list (`initialise`), the iterate for each time-step (`solve`) and then perform the threshold edge-detection (`threshold`):

```
> solveDiffusion :: (Fractional a, Num b, Ord a) => a -> Int -> a -> a -> [b]
> solveDiffusion v0 n lvl k
>  = map (threshold lvl 0) ( solve k v0 ( initialise n ) )
```

Initialising simply returns a list of `n` zeros:

```
> initialise :: Fractional a => Int -> [a]
> initialise n = replicate n 0.0
```

Solving iterates a time-step function (`timestep`):

```
> solve :: Num a => a -> a -> [a] -> [[a]]
> solve k v0 = iterate (timestep k v0)
```

Thresholding simply searches for the first entry less than or equal to the threshold, and returns the index of that point:

```
> threshold :: (Ord a, Num b) => a -> b -> [a] -> b
> threshold lvl i [] = i
> threshold lvl i (x:xs)
>   | x <= lvl  = i
>   | otherwise = threshold lvl (i+1) xs
```

The time-step calculation is most complex, as the ith element of the output list depends on the $i-1$th, ith and $i+1$th members of the input list, with special calculations at the boundaries.

```
> timestep :: Num a => a -> a -> [a] -> [a]
> timestep k v0 (seg1:rest@(seg2:segs))
>  = (step k v0 seg1 seg2):(timestep2 k seg1 rest)

> timestep2 k segminus [seg,segplus]
>  = [step k segminus seg segplus, endstep k seg segplus]
> timestep2 k segminus (seg:rest@(segplus:segs))
>  = (step k segminus seg segplus):(timestep2 k seg rest)
```

The `step` function implements

$$v_{(i,t+1)} = v_{(i,t)} + k \cdot (v_{(i-1,t)} - 2 \cdot v_{(i,t)} + v_{(i+1,t)})$$

```
> step :: Num a => a -> a -> a -> a -> a
> step k vminus v vplus = v + k*(vminus - 2*v + vplus)
```

The `endstep` function implements

$$v_{(i,t+1)} = v_{(i,t)} + k \cdot (v_{(i-1,t)} - v_{(i,t)})$$

```
> endstep :: Num a => a -> a -> a -> a
> endstep k vminus v = v + k*(vminus - v)
```

If we run

```
take 100 (solveDiffusion 1.0 20 0.3 0.2)
```

we get

```
[0,0,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,
3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6,6,6,
6,6,6,6,6,6,6,6,6,6,6,6]
```

which gives us

```
x  t dt ddt

1  2
      7
2  9       4
     11
3 21       5
     16
4 37       5
     21
5 58       5
     26
6 84
```

Fitting this using `ddt=5` and `x=3,4` gives $t \approx 2.5x^2 - 1.5x + 3.0$. This predicts $t = 192$ for $x = 9$, and running the solver gets:

```
Experiments> take 192 (solveDiffusion 1.0 20 0.3 0.2)
[0,0,1,1,1,1,1, .... 8,8,8,9,9,9,9]
```

A slight overestimate !