3BA1-II: Numerical Methods

Trinity Term

Dr. Andrew Butterfield

http://www.cs.tcd.ie/Andrew.Butterfield/Teaching/3BA1/

Timetable:

- Monday, 10am, M21
- Thursday, 10am, M20
- Friday, 11am, M20

Solving Quadratics (I)

Consider the following quadratic equation ($a = 1$):

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

The following is a way to solve such quadratics:

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

So for example, the roots of $x^2 - 3x + 2$ are 1 and 2 respectively.

How might we solve this by computer?

```
solvequad :: (Double,Double) -> (Double,Double)
solvequad (b,c) = (r1,r2) where ...
```

Solving Quadratics (II)

The following is a way to solve such quadratics:

```
solvequad :: (Double,Double) -> (Double,Double)
solvequad (b,c) = (r1,r2) where
  b' = -b/2
  d = b'*b'-c
  e = sqrt d
  e' = if b < 0.0 then -e else e
  r1 = b'+e
  r2 = b'-e

Experiments> solvequad (-3.0,2.0)
(2.0,1.0) :: (Double,Double)
```

How can we test this ?

Making Quadratics

Given two roots, r_1 and r_2, we can derive a quadratic with those roots by

$$\begin{aligned} 0 &= (x - r_1)(x - r_2) \\ &= x^2 - (r_1 + r_2)x + r_1 r_2 \\ &= x^2 + bx + c \end{aligned}$$

where

$$\begin{aligned} b &= -(r_1 + r_2) \\ c &= r_1 r_2 \end{aligned}$$

```
mkquad :: (Double,Double) -> (Double,Double)
mkquad (r1,r2) = (-(r1+r2),r1*r2)

Experiments> mkquad (2.0,1.0)
(-3.0,2.0) :: (Double,Double)
```

Some examples (I)

Try a quadratic with roots 1 and 10:

```
Experiments> mkquad (10,1)
(-11.0,10.0) :: (Double,Double)
Experiments> solvequad (-11.0,10.0)
(10.0,1.0) :: (Double,Double)
```

Looks OK

Some examples (II)

Try roots 1 and 10^{-8}:

```
Experiments> mkquad (1,1.0e-8)
(-1.0,1.0e-008) :: (Double,Double)

Experiments> solvequad (1.0,2.98023e-008)
(1.0,2.98023e-008) :: (Double,Double)
```

The answer is almost 3 times too large!

Some examples (III)

Try roots 1 and 10^{-9}:

```
Experiments> mkquad (1,1.0e-9)
(-1.0,1.0e-009) :: (Double,Double)
Experiments> solvequad (-1.0,1.0e-009)
(1.0,0.0) :: (Double,Double)
```

One root is zero!

What is going wrong?

Repeated Summing (I)

Let us try lots of additions

$$\sum_{i=1}^{100,000} 0.1 = 10,000$$

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

Repeated Summing (II)

Add $\frac{1}{10}$ a hundred thousand times to get ten thousand:

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

The correct answer is 10,000 — why the (small) error?

Repeated Summing (III)

Try adding $\frac{1}{100}$ a million times to get 10,000:

```
Experiments> ksum 0.01 1000000
9865.22 :: Double
```

Why the (large) error?

Addition is Associative

It is well known that

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

The following program tests this law:

```
> test_assoc a b c
>  = ((x,y),(y-x,x==y))
>  where
>   x = a+(b+c)
>   y = (a+b)+c
```

It takes in a, b and c, and returns the two sums, their difference, and a boolean indicating if they are equal.

Testing Associativity (I)

Let's try $a = 1$ with $b = 2$ and $c = 3$:

```
Experiments> test_assoc 1.0 2.0 3.0
((6.0,6.0),(0.0,True))
```

This look OK — Let us try another.

Testing Associativity (II)

Let's try $a = 1$ with $b = c = 3 \times 10^{-7}$:

```
Experiments> test_assoc 1.0 3.0e-7 3.0e-7
((1.0,1.0),(1.19209e-007,False))
```

The values print the same, but if fact differ, so we have seen that in this case

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

i.e.

$$1 + (3 \times 10^{-7} + 3 \times 10^{-7}) \neq (1 + 3 \times 10^{-7}) + 3 \times 10^{-7}$$

Summing Harmonic Series

Consider computing the following:

$$H(n) = \sum_{i=1}^{n} \frac{1}{i^2}$$

Summing $1 \ldots n$

We can accumulate the sum starting with $i = 1$ and working up to $i = n$:

```
hsumup n = hsum' n 1 0 where
 hsum' n i s
  | i > n      =  s
  | otherwise  =  hsum' n (i+1) $! (s+r*r)
  where r = 1/i
```


Summing $n \ldots 1$

We can accumulate the sum starting with $i = n$ and working down to $i = 1$:

```
hsumdn n = hsum' n 0 where
 hsum' n s
  | n<1  =  s
  | otherwise  =  hsum' (n-1) $! (s+r*r)
  where r = 1/n
```

Try $H(100,000)$

Let us try both functions:

```
Experiments> hsumup 100000
1.64473

Experiments> hsumdn 100000
1.64492
```

They differ!

Which one is more accurate?

What is going wrong?

We have tried some simple calculations using a computer.

We keep getting strange errors/results!

What is the reason for this?

How should we have done things differently?

Numerical Methods

Numerical Methods is the study of how to cope with the limitations of Computer Arithmetic, in order to get results in which we can be reasonably confident.

This is the topic of 3BA1 Part II

Lecture 1

This course concerns itself with:

- performing arithmetic with real numbers
- using computing machinery
- computer arithmetic involves finite approximations to infinitely precise quantities

Consider the various number types:

Naturals	$\mathbb{N}$	$0, 1, 2, 3, \ldots$
Integers	$\mathbb{Z}$	$\ldots, -2, -1, 0, 1, 2, \ldots$
Rationals	$\mathbb{Q}$	$0, \pm 1/1, \pm 1/2, \pm 2/1, \pm 1/3, \pm 3/1, \pm 2/3, \pm 3/2, \ldots$

Each of these contains an infinite number of numbers, but each number is itself *finite*.

However, the reals are different:

Reals $\mathbb{R}$

Each real number is itself an infinite object:

$$\begin{aligned} 1 &= 1.0000000000000000\ldots \\ 1/3 &= 0.33333333333333333\ldots \\ \pi &= 3.14159\ldots \\ e &= 2.717\ldots \end{aligned}$$

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 $-9, \ldots, +9$:

$$\pm d_0.d_1d_2d_3 \cdot 10^e$$

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 $a + b$:

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

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 $b + c$:

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

We now add a and $b + c$:

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

So for this floating point arithmetic scheme we get

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

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

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

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.

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.

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 $\bar{a} = a + \Delta a$ $\Delta a = -0.0002135\ldots$.

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

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

Relative error (Magnitude) : $\left|\frac{\Delta a}{a}\right|, \quad (a \neq 0)$ $\left|\frac{\Delta a}{a}\right| \leq 0.00016 \leq 0.0002$ (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.

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 $|R_B|$:

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

Why does error $= 0.0376$ get reported as < 0.04?

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.

Example (cont.)

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:

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

Key Definitions (II)

A number has t **correct decimals**, if $|\Delta a| \leq 0.5 \cdot 10^{-t}$.

Note a number only has (any) correct decimals if the error is ≤ 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
$312.538 \pm 0.5 \cdot 10^{-2}$	2	5
675000 ± 500		3

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:

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

Mean Value Theorem

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.

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 $1.43178210632763531544396012310 34\ldots$ so we can express our result as

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

(Common) Functions of two variables

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

Error propagation by Addition

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:

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

Error propagation by Subtraction

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

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

Error propagation by Multiplication

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:

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

Error propagation by Division (I)

For division:

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

Error propagation by Division (II)

For division:

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

Error Propagation Summary

So, for multiplicative operators we find 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.

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:

$$\begin{aligned} x_1 &= 10.123455 \pm 0.5 \cdot 10^{-6} \\ x_2 &= 10.123789 \pm 0.5 \cdot 10^{-6} \\ \left|\frac{\Delta x_i}{x_i}\right| &\leq \pm 0.5 \cdot 10^{-7} \\ y &= x_1 - x_2 \\ &= -0.000334 \pm 1 \cdot 10^{-6} \\ \frac{|\Delta y|}{y} &\leq \frac{10^{-6}}{0.000334} < 3 \cdot 10^{-3} \end{aligned}$$

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

By *"underlying 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.

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

$$x_1 = 5.55731 \cdot 10^{-2} \pm 0.5 \cdot 10^{-7}$$

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:

$$b \leq 0: \quad x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-2c}{b - \sqrt{b^2 - 4ac}}$$

$$b \geq 0: \quad x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

In each case, we are avoiding *underlying subtraction* of b and $\sqrt{b^2 - 4ac}$.

In fact, the calculation involving these quantities is always the same for both solutions!

Representation of Numbers in Computers

Decimal number system is a **position system** with **base 10**.

Let β be natural number greater than 1

$$\begin{aligned} \beta &: \mathbb{N} \\ \beta &\geq 2 \end{aligned}$$

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 $\beta - 1$:

$$\begin{aligned} d_i &: \mathbb{N} \\ 0 \leq d_i &< \beta \end{aligned}$$

The value of such a number is

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

However, real processors used fixed world length (typically 32 or 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 numbers get smaller.

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

Floating Point Representations (Infinite)

$$
\begin{aligned}
X &= M \cdot \beta^e \\
\beta & \qquad\qquad\qquad\qquad\qquad\qquad \text{(Base)} \\
e & \qquad\qquad\qquad\qquad\qquad\qquad \text{(Exponent)} \\
M &= \pm D_0.D_1D_2D_3\ldots \qquad \text{(Mantissa)} \\
0 \leq D_i &< \beta \\
D_0 &\neq 0
\end{aligned}
$$

Floating Point Representations (Finite)

$$
\begin{aligned}
x &= m \cdot \beta^e \\
\beta &: \mathbb{N}, \quad \beta > 1 \qquad \text{(Base)} \\
e &: \mathbb{Z} \qquad\qquad\qquad\qquad\qquad\qquad \text{(Exponent)} \\
m &= \pm d_0.d_1d_2\ldots d_t \\
0 \leq d_i &< \beta \\
d_0 &\neq 0
\end{aligned}
$$

Here m is M rounded to $t+1$ digits (t after the point, one before) — $|\Delta m| \leq \frac{1}{2}\beta^{-t}$

We *normalise*, so $1 \leq |m| < \beta$.

We give limits to e: $L \leq e \leq U$

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

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

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

$$\begin{aligned} m &= \pm d_0.d_1 d_2 \ldots d_t \\ 0 \leq d_i &< \beta \\ d_0 &\neq 0 \\ 1 \leq |m| &< \beta \\ L \leq e &\leq U \\ x &= m \cdot \beta^e \end{aligned}$$

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

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.

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

The *relative rounding error* is estimated as

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

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

All (normalised) numbers in floating point representations have a relative accuracy accuracy bounded above by the unit-roundoff, regardless of the size of the number.

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.

input f.p. numbers (β, t, L, U)

$\downarrow$

expand $t + 1$ digits to ?? digits ;

$\downarrow$

compute

$\downarrow$

round result to $t + 1$ digits

$\downarrow$

output f.p. number (β, t, L, U)

(??) 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 $(+, -, \times, /)$ 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

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

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):

$$
\begin{aligned}
&\mathit{Add}/\mathit{Sub}(m_x, e_x, m_y, e_y) \quad \widehat{=} \\
&\quad e_z := e_x; \\
&\quad \textbf{if} \;\; e_x - e_y \geq t + 3 \\
&\quad \textbf{then} \;\; m_z := m_x \\
&\quad \textbf{else} \\
&\qquad m_y := \text{shift } m_y \; (e_x - e_y) \text{ digits to the right}; \\
&\qquad m_z := m_x \pm m_y \\
&\quad \textbf{endif}; \\
&\quad \mathit{TidyUp}(m_z, e_z)
\end{aligned}
$$

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

Floating Point Multiplication

Multiplication is very straightforward:

$$
\begin{aligned}
&\mathit{Mul}(m_x, e_x, m_y, e_y) \\
&\widehat{=} \\
&\quad e_z := e_x + e_y; \\
&\quad m_z := m_x \times m_y; \\
&\quad \mathit{TidyUp}(m_z, e_z)
\end{aligned}
$$

Floating Point Division

So is division:

$$
\begin{aligned}
&\mathit{Div}(m_x, e_x, m_y, e_y) \\
&\widehat{=} \\
&\quad e_z := e_x - e_y; \\
&\quad m_z := m_x \,/\, m_y; \\
&\quad \mathit{TidyUp}(m_z, e_z)
\end{aligned}
$$

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)

Finalise

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 infinite).

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.

The IEEE Standard allows un-normalised numbers in this situation.

Key Result re Arithmetic Error

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

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

Here $\mathrm{fl}[x \odot y]$ denotes the approximate result of the floating point implementation of $x \odot y$.

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.

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:

$$\begin{aligned}
\hat{S}_1 &= x_1 && \text{(strictly } x_1(1+\epsilon_1)\text{, but } \epsilon_1 = 0\text{)} \\
\hat{S}_2 &= (\hat{S}_1 + x_2)(1+\epsilon_2) \\
&= (x_1 + x_2)(1+\epsilon_2) \\
\hat{S}_3 &= (\hat{S}_2 + x_3)(1+\epsilon_3) \\
&= ((x_1 + x_2)(1+\epsilon_2) + x_3)(1+\epsilon_3) \\
&= x_1(1+\epsilon_2)(1+\epsilon_3) + x_2(1+\epsilon_2)(1+\epsilon_3) + x_3(1+\epsilon_3) \\
\hat{S}_4 &= (\hat{S}_3 + x_4)(1+\epsilon_4) \\
&= ((x_1(1+\epsilon_2)(1+\epsilon_3) + x_2(1+\epsilon_2)(1+\epsilon_3) + x_3(1+\epsilon_3)) + x_4)(1+\epsilon_4) \\
&= x_1(1+\epsilon_2)(1+\epsilon_3)(1+\epsilon_4) \\
&+ x_2(1+\epsilon_2)(1+\epsilon_3)(1+\epsilon_4) \\
&+ x_3(1+\epsilon_3)(1+\epsilon_4) \\
&+ x_4(1+\epsilon_4) \\
&\vdots
\end{aligned}$$

The general case:

$$\begin{aligned}
\hat{S}_n &= (\hat{S}_{n-1} + x_n)(1+\epsilon_n) \\
&= x_1(1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)\cdots(1+\epsilon_n) \\
&+ x_2(1+\epsilon_2)(1+\epsilon_3)\cdots(1+\epsilon_n) \\
&+ x_3(1+\epsilon_3)\cdots(1+\epsilon_n) \\
&\vdots \\
&+ x_i(1+\epsilon_i)\cdots(1+\epsilon_n) \\
&\vdots \\
&+ x_n(1+\epsilon_n)
\end{aligned}$$

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.

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

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.

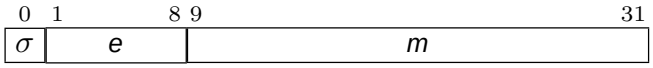
Basically the standard precisely defines two main formats — *single and double precision floating point*, and gives somewhat looser specifications for two extended versions of these formats.

The standard mandates that the operations $+, -, \times, /$ and $\sqrt{}$ 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	∞
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.

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 $\frac{n^2+3n}{2}$ 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 (called 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 $S = x_0 + x_1 + x_2 + \cdots$.

Alternating Series

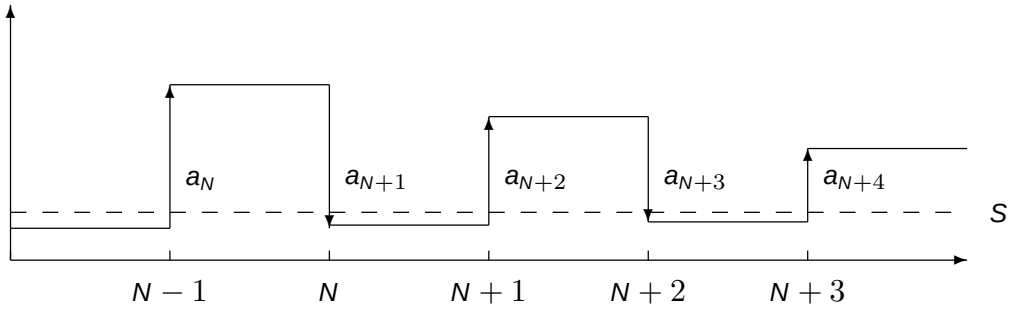
Consider a series where each successive terms has opposite sign and is smaller in magnitude than its predecessor:

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

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 (i)

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

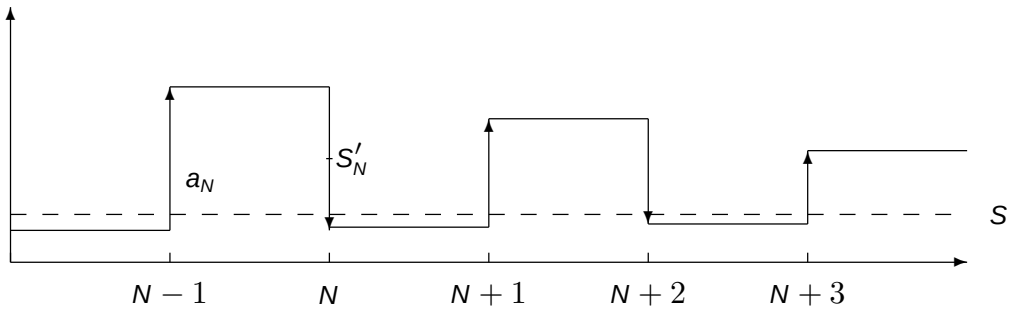
We want $|R_N| \leq \pm 0.5 \cdot 10^{-3}$, so we calculate

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

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

Example (ii)

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.

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:

$$S = \sum_0^\infty a_n = \sum_0^\infty f(n)\Delta x \text{ where } \Delta x = 1$$

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 $3 \cdot 10^{-6}$

An efficient implementation of $\sqrt{}$

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

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

- Noting 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,
- In each case we have a binary number with 2 digits to the left of the point:

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

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

$$01.00\ldots00 \quad \text{to} \quad 11.11\ldots10$$

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 the 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 ($01.00 \ldots 11.11$):

$$\begin{aligned} 01.00_2 &\mapsto \sqrt{1.00_{10}} \\ 01.01_2 &\mapsto \sqrt{1.25_{10}} \\ \vdots \quad &\mapsto \\ 11.10_2 &\mapsto \sqrt{3.50_{10}} \\ 11.11_2 &\mapsto \sqrt{3.75_{10}} \end{aligned}$$

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 (i)

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

$$\begin{aligned} & x_{n+1} - \sqrt{a} \\ = & \quad \text{“ defn. } x_{n+1} \text{ ”} \\ & \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) - \sqrt{a} \\ = & \quad \text{“ place all over } 2x_n \text{ ”} \\ & \frac{x_n^2 + a - 2x_n\sqrt{a}}{2x_n} \\ = & \quad \text{“ rearrange ”} \\ & \frac{x_n^2 - 2x_n\sqrt{a} + a}{2x_n} \end{aligned}$$

Proof (ii)

$$
\begin{aligned}
& \frac{x_n^2 - 2x_n\sqrt{a} + a}{2x_n} \\
= \quad & \text{“ } (z-b)^2 = z^2 - 2zb + b^2 \text{ with } z = x_n \text{ and } b = \sqrt{a} \text{ ”} \\
& \frac{1}{2x_n}(x_n - \sqrt{a})^2 \\
\geq \quad & \text{“ } x_n \text{ positive implies expression is non-zero and positive ”} \\
& 0 \\
\Rightarrow \quad & \text{“ } x - y \geq 0 \text{ means that } x \geq y \text{ ”} \\
& x_{n+1} \geq \sqrt{a}
\end{aligned}
$$

Proof (iii)

We now show that the sequence is decreasing:

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

Error Estimate for $\sqrt{a}$

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:

$$\left|x_0 - \sqrt{a}\right| \approx 2^{-4}$$

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:

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

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

Exercise

If we use 6 bits to index the table:

1. how many entries do we need ?
2. how many iterations do we need to get a sufficiently accurate result ?

Answer

If we use 6 bits to index the table:

1. we need 48 entries
2. we only need two iterations.

Tutorial 1 — Exercise 1

Consider the following data: $a = 1.234 \cdot 10^{3} \pm 0.5 \cdot 10^{-0}$ and $b = 6.789 \cdot 10^{-2} \pm 0.5 \cdot 10^{-5}$
Estimate the absolute and relative errors of the following calculations (assuming that the implementation of the arithmetic operations is fully accurate):

1. $a + b$
2. $a - b$
3. $b - a$
4. $a \cdot b$
5. $a \,/\, b$

Additive operators add absolute error magnitudes.

Multiplicative operators add relative error magnitudes.

Tutorial 1 — Exercise 2

Estimate the error in computing the following series out to four terms:

1.

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

where $x = 0.1$

2.

$$S = \sum_{k=1}^{\infty} \frac{(-1)^k}{k^3}$$

If series is positive, but strictly decreasing, with function f such that $f(n) = a_n$, then $|R_n| = -F(n+1)$ where $F = \int f(x)dx$.

If series is alternating and decreasing, $|R_n| \leq |a_{n+1}|$

Tutorial 1 — Exercise 3

How many terms of the following series do we need to compute in order to get an accuracy of $\pm 0.5 \cdot 10^{-6}$?

$$\sum_{n=1}^{\infty} \frac{1+n}{n^3}$$

If series is positive, but strictly decreasing, with function f such that $f(n) = a_n$, then $|R_n| = -F(n+1)$ where $F = \int f(x)dx$.

If series is alternating and decreasing, $|R_n| \leq |a_{n+1}|$

Tutorial 1 — Exercise 4

If we use a lookup table for $\sqrt{x}$ based on the first two bits of the value $1 \leq a < 4$, what size table do we require, and how many iterations of Newton-Raphson are required to get to a precision of 2^{-24}?

Note that:

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

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

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

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* (moving away from the solution).

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

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

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 $0.5 < x^* < 0.6$.

Interval Techniques

Interval techniques bracket the solution between lower and upper limits which then get moved in incrementally closer to the solution.

Bisection

Given an initial interval, we determine subsequent intervals by:

1. finding the mid-point
2. evaluating f at that point
3. 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 $[x_{n-1}, x_n]$, $\text{sign}(f(x_{n-1})) \neq \text{sign}(f(x_n))$ we compute

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

If $\text{sign}(f(x_{n+1})) = \text{sign}(f(x_n))$ 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.

Bisection 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 $[x_{n-1}, x_n]$, $\text{sign}(f(x_{n-1})) \neq \text{sign}(f(x_n))$, 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:

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

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 techniques 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.

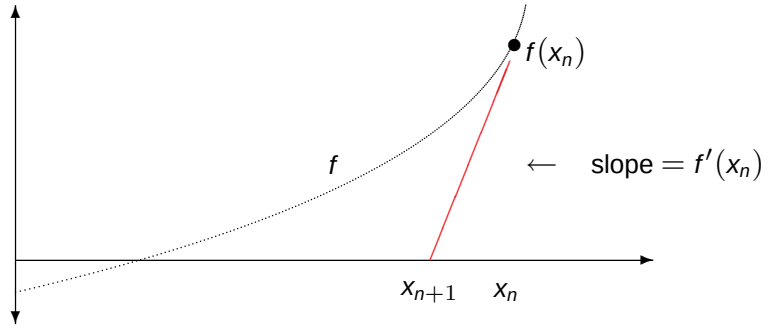
Point Techniques

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^*:

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

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

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

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

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

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

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

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.

The Big Question

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

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 ?

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 $x_n \longrightarrow x^*$ if and only if $\epsilon_n \longrightarrow 0$, as $n \longrightarrow \infty$.

We now develop the expression for ϵ_n:

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

Mean Value Theorem

$$f \in C[a,b] \ \wedge \ f' \textbf{ defined over } (a,b)$$

$$\Rightarrow$$

$$\exists\, c \in (a,b) \bullet f'(c) = \frac{f(b)-f(a)}{b-a}$$

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:

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

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 than 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):

$$\varphi_1(x) = e^{-x} \qquad \varphi_1'(x) = -e^{-x} \qquad |\varphi_1'(0.567)| \approx 0.567 < 1$$
$$\varphi_2(x) = -\log x \quad \varphi_2'(x) = -1/x \quad |\varphi_2'(0.567)| \approx 1/0.567 > 1$$

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.

Convergence for Newton-Raphson

Now we turn our attention to Newton-Raphson:

$$\varphi_3(x) = x - \frac{f(x)}{f'(x)}$$
$$\varphi_3'(x) = \frac{f(x)f''(x)}{(f'(x))^2}$$

The derivative of φ_3 is calculated as follows:

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

(cont. overleaf)

(cont. from prev.)

$$
\begin{aligned}
& 1 - \frac{(f'(x))^2}{(f'(x))^2} + \frac{f(x)f''(x)}{(f'(x))^2} \\
= & \quad \text{“ simplify ”} \\
& 1 - 1 + \frac{f(x)f''(x)}{(f'(x))^2} \\
= & \quad \text{“ simplify ”} \\
& \frac{f(x)f''(x)}{(f'(x))^2}
\end{aligned}
$$

We compute at the fixed point:

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

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” ?

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:

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

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

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

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*.

$$|\epsilon_{n+1}| \leq k\,|\epsilon_n|^2$$

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.

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 off quickly and accurately.

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:

$$f'(x) \approx D_0(h) = \frac{1}{2}(D_+(h) + D_-(h)) = \frac{f(x+h) - f(x-h)}{2h}$$

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

For second derivatives, we have the definition:

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

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:

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

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:

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

We have that

$$R_T = D_+(h) - f'(x) = \frac{1}{2}f''(\xi)h$$

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 $O(h)$.

Note we can say $D_+(h) = f'(x) + R_T$, or in general,

$$D_+(h) = f'(x) \pm |R_T|$$

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:

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

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

$$D_0(h) = f'(x) + b_1 h^2 + b_2 h^4 + \cdots$$

where we 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 $f'(1) = e^1 \approx 2.71828\ldots$, 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 a 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 initially by about a factor of 100 each time.
- However, at $7.5 \cdot 10^{-4}(!)$, where the shrinkage rate drops by half.
- It gets worse at the next drop in h, because the error actually gets larger (!!)
- This gets worse as h shrinks further , until we see the error for the smallest h is almost 50% of the true value (!!!).

What we are observing is the limit imposed by the unit roundoff error, i.e. the best precision available from the floating point number system in use.

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 has a subtraction which is safe because we are subtracting a small quantity (c_n) from a large one.

Rounding Error in Difference Equations

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 (well-behaved) f that $f(x+h)$ 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:

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

The calculation error is then given as

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

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.

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

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

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.

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*

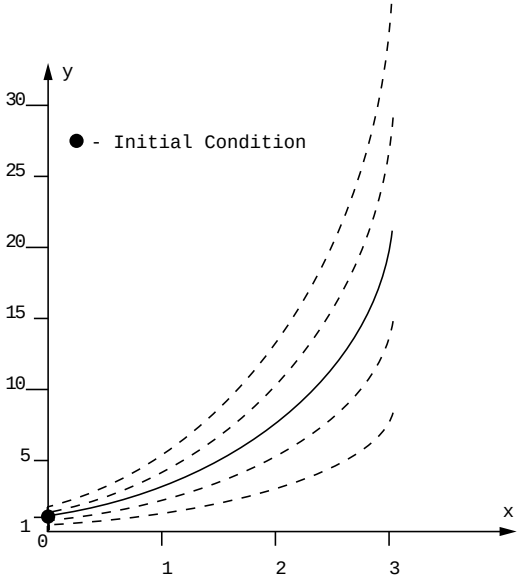
$$y(a) = \alpha$$

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 $y(0) = 1$.

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 $L = 1$.

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:

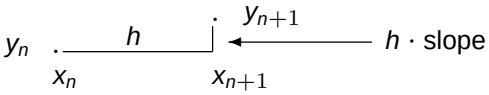
$$x_0 = a \qquad y_0 = y(x_0) = y(a) = \alpha$$

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

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

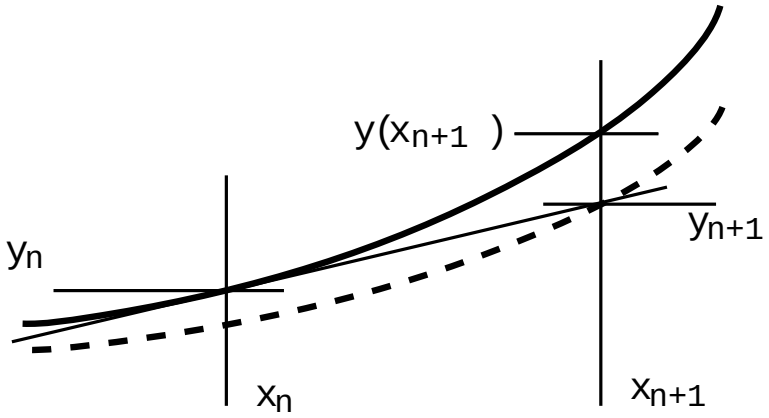
At each point after x_0, we accumulate an error, because we are using the slope at x_n to estimate y_{n+1}, which assumes that the slope doesn't change over interval $[x_n, x_{n+1}]$.

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:

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

We then look at y_{n+1}:

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

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.

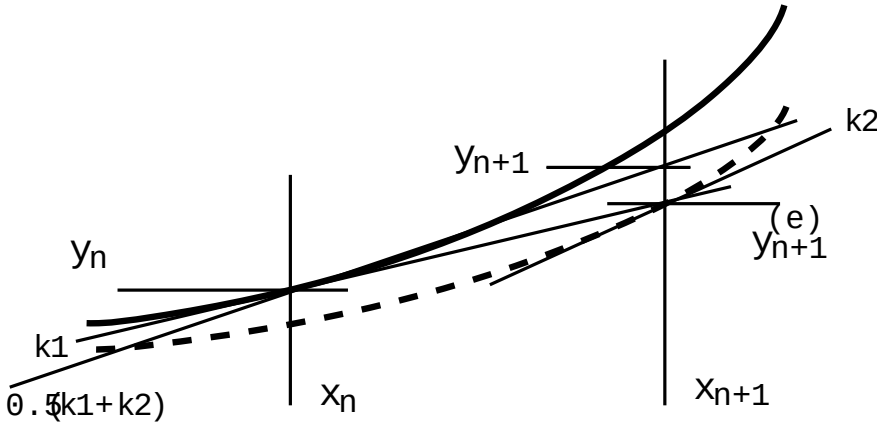
So we see that the local truncation error for Euler's Method is $O(h^2)$.

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 $O(h^p)$.

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 $y_{n+1}^{(e)}$ 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 $y'(x_{n+1}, y_{n+1}^{(e)}) = f(x_{n+1}, y_{n+1}^{(e)})$ 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:

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

We then compute y_{n+1} as

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

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 evaluate 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 $O(h^4)$.

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 need a polynomial of degree n to fit data with $n + 1$ points. It also becomes clear that with polynomials of higher degree than this, there are 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 one 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

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

Case $n + 1 = 2$ (cont.)

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:

$$C_2 = \frac{f_3 - f_1}{(x_3 - x_1)(x_3 - x_2)} - \frac{f_2 - f_1}{(x_2 - x_1)(x_3 - x_2)}$$

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

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

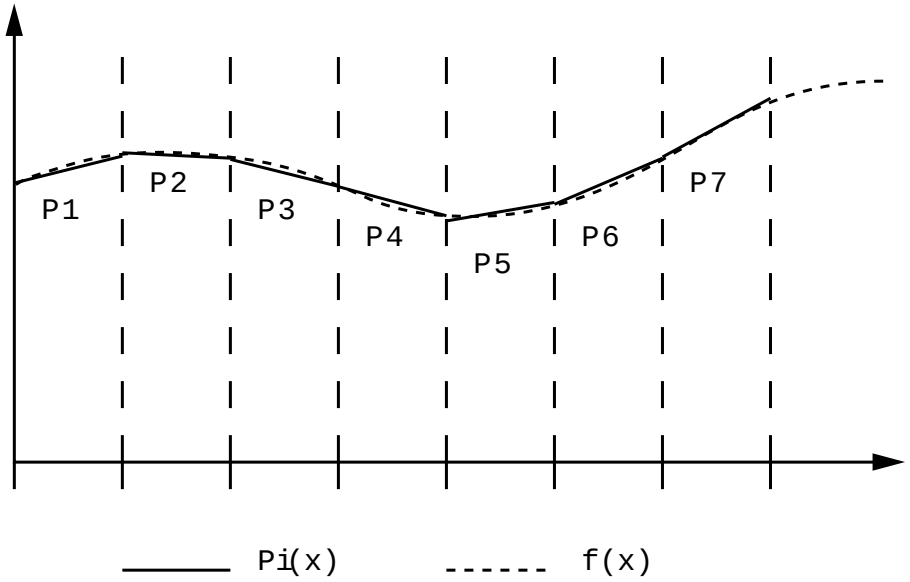
on the interval $[-1, +1]$ has the surprising property, that as the degree of the interpolating polynomial rises, so does the error at points in between the interpolation points. In fact he found that the error tends to infinity as $n \to \infty$.

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 $P1(x)$, $P2(x)$, ... $Pn(x)$, each of degree 1.



Numerical Integration

Remember integrating f means finding F such that:

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

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:

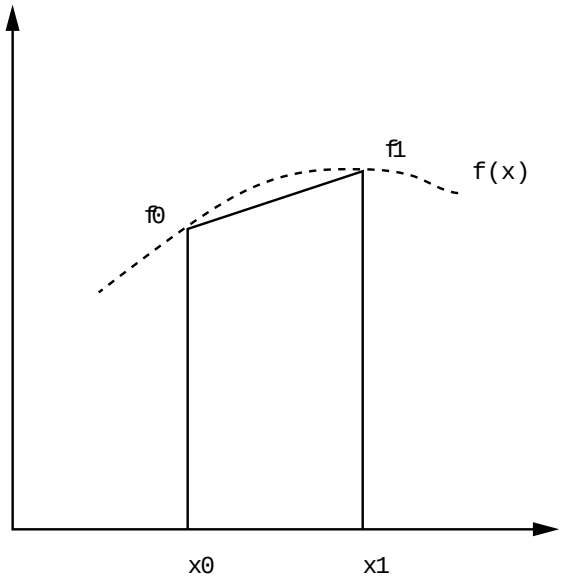
$$f(x) = \frac{1}{1+25x^2}$$

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

In practise we restrict k to 1 or 2.

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:

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

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:

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

$$
\begin{aligned}
& \frac{h^3 f''(\eta)}{2} \left(\frac{p^3}{3} - \frac{p^2}{2}\right)\Big|_0^1 \\
= \quad & \text{“ evaluate ”} \\
& \frac{h^3 f''(\eta)}{2}\left(\frac{1}{3} - \frac{1}{2}\right) \\
= \quad & \text{“ simplify ”} \\
& \frac{h^3 f''(\eta)}{12}
\end{aligned}
$$

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 !

Simpson's Rule ($k = 2$)

Simpson's rule uses a parabola (polynomial of degree 2) to fit three points:

The resulting formula obtained is:

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

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.

Solving The Diffusion Equation

Recall that from 3BA4, when analysing signal flow, we came up with the following 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:

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

This is the Diffusion Equation, which has no analytic solution.

We are now in a position to solve this numerically, and to backup the assertion made in 3BA4 that propagation along a thin wire is $O(\ell^2)$.

We split the line of length ℓinto n segments, each of length Δx, so that $\ell = n\Delta x$. We let $x_i = i\Delta x$ denote the ith segment.

We choose a time interval of Δt, and define our unit of time in terms of this, so $t+1$ denotes the time slot Δt seconds after time t.

We denote the voltage at x_i at time t by $V(i,t)$.

How do we transform our differential equation to work with this scheme ?

We shall use Forward difference for $\partial V/\partial t$ and the (symmetric) difference equation for $\partial^2 V/\partial x^2$:

$$rc\frac{V(i,t+1)-V(i,t)}{\Delta t} = \frac{V(i+1,t)-2V(i,t)+V(i-1,t)}{\Delta x^2}$$

Rearrange, keeping all terms at time t together:

$$V(i,t+1) = V(i,t) + \frac{\Delta t}{rc\Delta x^2}(V(i+1,t)-2V(i,t)+V(i-1,t))$$

This is our finite approximation to the differential equation.

We note that we have the voltage at point i during the next time step ($t+1$) expressed solely in terms of voltages at this an neighbouring point during this time-step (t).

Hence, we can start at $t = 0$ and compute values successively for $t = 1, 2, \ldots$.

We want to solve this, subject to the boundary conditions:

- Left end held at 5V

$$V(0,t) = 5V$$

- Wire initially at 0V (except at extreme left)

$$V(i,0) = 0V, \qquad i > 0$$

How do we implement this ?

What data structures should we use ?

Data-Structures

We can represent the state (voltage) of the line at time t by an array indexed 0 to n, the i th entry corresponding to $V(i,t)$.

We could then have an array of these, indexed by time, from 0 upwards, to some limit.

Assume we have an array `v[0..n,0..tmax]`

How do we initialise the system ?

Initialising

We simply set:

Left Boundary `v[0,0] = 5.0`

Initial Wire State `v[i,0] = 0.0`, for `i` in range $1 \ldots n$

How do we compute the next time step ?

```
v[i,t+1] = ?
```


Computing the next time step

Given that we have computed `v(i,t)`, for all `i` in range $1 \ldots n$, we compute for the next time-slot using the following formula:

```
v[i,t+1] = v[i,t] + k*(v[i+1,t]-2*v[i,t]+v[i-1,t])
```

where `k` is the pre-computed value of $\frac{\Delta t}{rc\Delta x^2}$.

How do we handle the extreme ends of the line ($i = 0$ or n) ?

Handling line ends

For $i = 0$, the solution is simple: this value does not change:

```
v[0,t+1] = v[0,t]
```

For $i = n$, we need to be careful — there is no `v[n+1,t]`, so what do we use in its stead?

Simply treat `v[n+1,..]` as zero ?

```
v[n,t+1] = v[i,t] + k*(v[i-1,t]-2*v[i,t])
```

Is this OK ?

Be very very careful . . .

Setting `v[n+1,..]` to zero has the effect of modelling the situation where $0V$ volts is applied to the left of the wire.

This is not the problem we were asked to solve !

The correct answer is to set `v[n+1,t]` equal to `v[n,t]`, so it effectively “floats”.

This can also be justified by deriving the appropriate equation for the righthand end, in the manner used in 3BA4.

The correct equation for the rightmost location is therefore:

```
v[n,t+1] = v[n,t] + k*(v[n-1,t]-v[n,t])
```

Putting it all together

A complete program might look like the following (in a vaguely C-like language) :

```
solveDiffEqn(r,c,l,vleft:double) {
  double  v[N+1,TMAX+1], k;
  int i,t;

  k = .. some appropriate calculation .. (see later)
  v[0,0] = vleft;
  for(i=1;i<=N;i++) v[i,0]=0.0;
  for(t=0;t<TMAX;t++){
     v[0,t+1] = v[0,t];
     for(i=1;i<N;i++)
       v[i,t+1] = v[i,t] + k*(v[i+1,t]-2*v[i,t]+v[i-1,t]) ;
     v[N,t+1] = v[N,t] + k*(v[N-1,t]-v[N,t])
  } ;
  ... print out array values ...
}
```


How do we choose N, TMAX and k ?

Choosing N, TMAX and k

We don't have a theory (in 3BA1 at least) to help us with error estimates: we shall therefore experiment.

First, we simplify things by fixing N and TMAX, to 20 and 1000.

The choice of TMAX is not very important and can be adjusted to suit.

Having fixed N, however, we have made a choice of Δx:

$$\Delta x = \frac{\ell}{N}$$

Given that we have r, c and ℓas parameters, and that we have no fixed Δx, then our choice of Δt will determine k:

$$k = \frac{1}{rc\Delta x^2} \cdot \Delta t$$

So we now concentrate on our choice of k — we can work out Δt from this.

What is the best choice of k ?

Spreadsheet Experiments

The algorithm just described can actually be entered into a spreadsheet !

This makes is very easy to experiment

Choosing the best **k**

We find that k greater that 0.5 leads to strange unstable results

There is no advantage to very small k, as this leads to slow progress in the simulation.

We choose k = 0.5 as the best value to get some useful data.

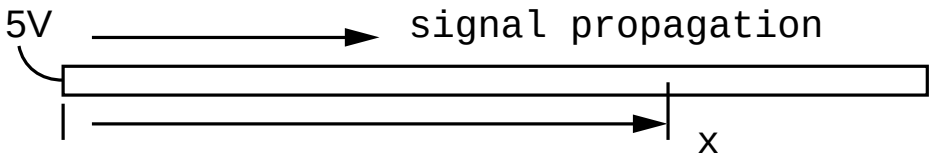
Is propagation time $O(\ell^2)$?

Experimental Goal

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

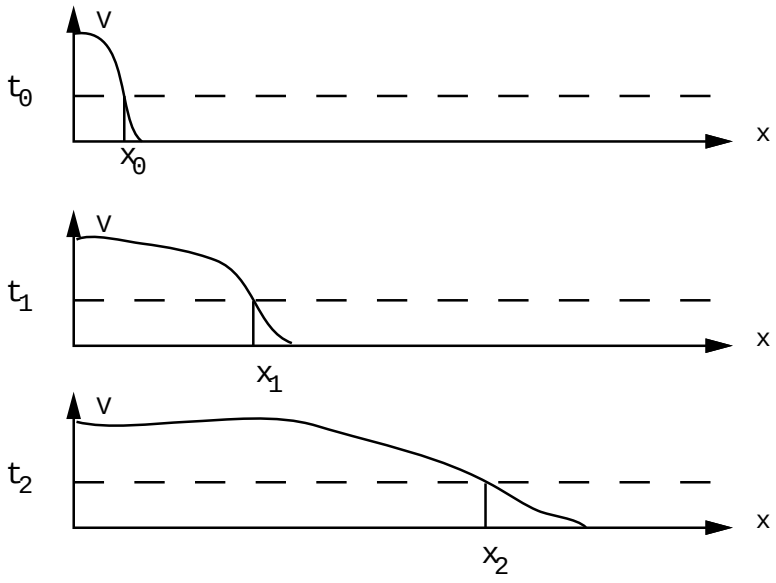
For any experiment, it is important to know in advance what outcome to expect.

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



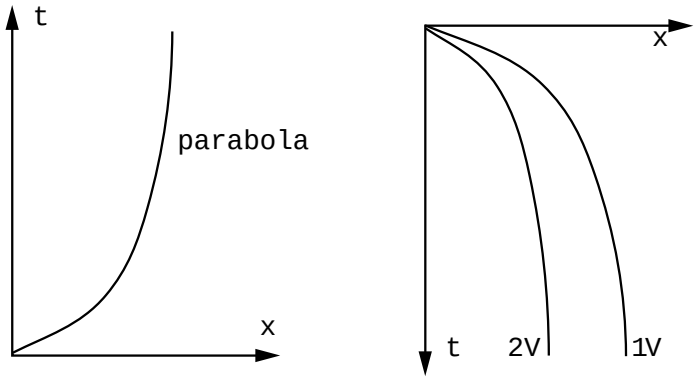
We are hoping to see that the time it takes for a voltage of a certain threshhold value to reach point x is proportional to x^2.

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 last location along the wire where the voltage is greater than the threshold

The following slide 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 *emphasis*, 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

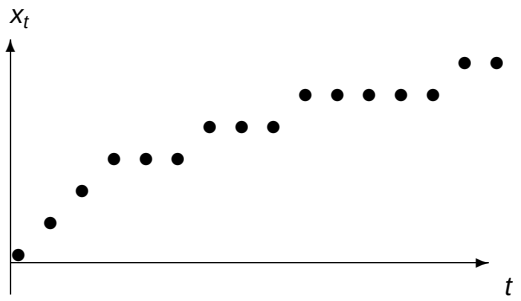
Analysis

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	0	1	2	3	3	3	4	4	4	5	5	5	5	5	6	6

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	0	1	4	9	9	9	16	16	16	25	25	25	25	25	36	36

This looks like a reasonably good straight-line fit

Detailed Analysis

It would be possible to apply curve-fitting techniques to try to verify the $O(\ell^2)$ relationship

However, the (t, x_t) data we have is not itself fully accurate, and this makes detailed analysis a little tricky.

For now, we can assume that we have verified our conjecture satisfactorily.

Back to reality

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, what would our values of Δx and Δt be ?

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

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

Exam

- 8 questions, 6 in Section A (Stats), 2 in Section B (Num. Analysis)
- You must answer at least one section B question.
- Past papers:
  - 2004 paper is also valid.
  - Q7s from 1999 to 2003 are relevant
  - Ignore Q8s
  - Ignore 1998 completely.

And that’s all folks!