COMP20005 Engineering Computation

Additional Notes

Numeric Computation, Part A

Lecture slides © Alistair Moffat, 2013

Algorithmic objectives

For symbolic processing (for example, sorting strings), desire algorithms that are:

- Above all else, correct
- Straightforward to implement
- Efficient in terms of memory and time
- (For massive data) Scalable and/or parallelizable
- (For simulations) Statistical confidence in answers and in the assumptions made.

Algorithmic objectives

For numeric processing, desire algorithms that are:

- Above all else, correct
- Straightforward to implement
- Effective, in that yield correct answers and have broad applicability and/or limited restrictions on use
- Efficient in terms of memory and time
- (For approximations) Stable and reliable in terms of the underlying arithmetic being performed.

The last one can be critically important.

Example 1

Wish to compute

$$f(x) = x \cdot \left(\sqrt{x+1} - \sqrt{x}\right)$$

and

$$g(x) = \frac{x}{\sqrt{x+1} + \sqrt{x}}.$$

So write the obvious functions, and print some values...

- `sqdiff.c`

Hmmmm, why did that happen?

Example 2

Wish to compute

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

So write the obvious function, and print some values...

- `logsum.c`

Hmmmm, why did that happen?

Pitfalls

In all numeric computations need to watch out for:

- subtracting numbers that are (or may be) close together, because absolute errors are additive, and relative errors are magnified
- adding large sets of small numbers to large numbers one by one, because precision is likely to be lost
- comparing values which are the result of floating point arithmetic, zero may not be zero.

And even when these dangers are avoided, numerical analysis may be required to demonstrate the convergence and/or stability of any algorithmic method.

Numbers – bits, bytes, and words

Inside the computer, everything is stored as a sequence of binary digits, or bits.

Each bit can take one of two values – "0", or "1".

A byte is a unit of eight bits, and most computers a word is a unit of either four or eight bytes.

A word typically stores a set of 32 or 64 bits. The interpretation of that bit sequence depends on the type of the variable involved, and the representation used for the different data types.

Variables and sizes

The preprocessor "function" `sizeof()` can be supplied with either a type or a variable, and at compilation time is replaced by the number of bytes occupied by that type:.

- `sizeof.c`

Binary numbers

In `char`, `short`, `int`, and `long` variables, the bits are used to create a binary number.

In decimal, the number 345 describes the calculation $3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$.

Similarly, in binary, the number 1101 describes the computation $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, or thirteen in decimal.

Binary numbers

Binary counting: 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, and so on.

With a little bit of practice, you can count to 1,023 on your fingers; and with a big bit of practice, to 1,048,575 if you use your toes as well.

There are two further issues to be considered:

- negative numbers, and
- the fixed number of bits w in each word.

Integer representations

In an unsigned $w = 4$ bit system, the biggest value than can be stored is 1111, or 15 in decimal.

Adding one then causes an integer overflow, and the result 0000.

Integer values are stored in fixed words, determined by the architecture of the hardware and design decisions embedded in the compiler.

Integer representations

The second column of Table 13.3 (page 232) shows the complete set of values associated with a $w = 4$ bit unsigned binary representation.

When $w = 32$, the largest value is $2^{32} - 1 = 4{,}294{,}967{,}295$.

Integer representations

Bit pattern	Integer representation		
	unsigned	sign-magn.	twos-comp.
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	−0	−8
1001	9	−1	−7
1010	10	−2	−6
1011	11	−3	−5
1100	12	−4	−4
1101	13	−5	−3
1110	14	−6	−2
1111	15	−7	−1

Sign-magnitude representation

To handle negative numbers, one bit could be reserved for a sign, and $w - 1$ bits used for the magnitude of the number.

The third column of Table 13.3 shows this sign-magnitude interpretation of the 16 possible $w = 4$-bit combinations.

There are two representations of the number zero.

Adding one to `INT_MAX` gives -0.

Twos-complement representation

The final column of Table 13.3 shows twos-complement representation. In it, the leading bit has a weight of $-(2^{w-1})$, rather than 2^{w-1}.

If that bit is on, and $w = 4$, then subtract $2^3 = 8$ from the unsigned value of the final three bits.

So 1101 is expanded as
$1 \times -(2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, which is minus three.

Twos-complement representation

The advantages of twos-complement representation are that

- there is only one representation for zero, and
- integer arithmetic is easy to perform.

For example, the difference $4 - 7$, or $4 + (-7)$, is worked out as $0100 + 1001 = 1101$, which is the correct answer of minus three.

Most computers use twos-complement representation for storing integer values.

Twos-complement representation

Revisiting an old program, it should now make more sense.

- `overflow.c`

Adding one the the biggest number in twos-complement gives the smallest number.

Twos-complement representation

On a $w = 32$-bit computer the range is from $-(2^{31}) = -2{,}147{,}483{,}648$ to $2^{31} - 1 = 2{,}147{,}483{,}647$. Beyond these extremes, `int` arithmetic wraps around and gives erroneous results.

If $w = 64$-bit arithmetic is used (type `long long`), the range is $-(2^{63})$ to $2^{63} - 1 = 9{,}223{,}372{,}036{,}854{,}775{,}807$, approximately plus and minus nine billion billion, or 9×10^{18}

The type `char` is also an integer type, and using 8 bits can store values from $-(2^7) = -128$ to $2^7 - 1 = 127$

Unsigned types

C offers a set of alternative integer representations, `unsigned char`, `unsigned short`, `unsigned int` (or just `unsigned`), `unsigned long`, and `unsigned long long`.

Negative numbers cannot be stored.

But will get printed out if you still use `"%d"` format descriptors. Use `"%u"` instead, or `"%lu"`, or `"%llu"`.

Unsigned types

C also provides low-level operations for isolating and setting individual bits in `int` and `unsigned` variables.

These operations include left-shift (<<), right-shift (>>), bitwise and (&), bitwise or (|), bitwise exclusive or (^), and complement (~).

There are some subtle differences between `int` and `unsigned` when bit shifting operations are carried out.

- `intbits.c`

Unsigned types

Table 13.5 (page 236) gives a final precedence table that includes all of the bit operations. If in doubt, over parenthesize.

Variables that are declared as `unsigned` types store positive values only, and can be used to manipulate raw bit strings.

Octal and hexadecimal

C also supports constants that are declared as octal (base 8) and hexadecimal (base 16) values. Beware! Any integer constant that starts with 0 is taken to be octal:

```
int o = 020;
int h = 0x20;
printf("o = %oo, %dd, %xx\n", o, o, o);
printf("h = %oo, %dd, %xx\n", h, h, h);
```

gives

```
o = 20o, 16d, 10x
h = 40o, 32d, 20x
```

Octal and hexadecimal

The standard Unix tool `bc` can be used to do radix conversions:

```
mac: bc
ibase=10
obase=2
25
11001
obase=8
25
31
obase=16
25
19
```

Floating point representations

The floating point types `float` and `double` are stored as:

- a one bit sign, then
- a w_e-bit integer exponent of 2 or 16, then
- a w_m-bit mantissa, normalized so that the leading binary (or sometimes hexadecimal) digit is non-zero.

Floating point representations

When $w = 32$, a `float` variable has around $w_m = 24$ bits of precision in the mantissa part. This corresponds to about 7 or 8 digits of decimal precision.

In a `double`, around $w_m = 48$ bits of precision are maintained in the mantissa part.

Floating point representations

For example, when $w = 16$, $w_s = 1$, $w_e = 3$, $w_m = 12$, the exponent is a binary numbers stored using w_e-bit twos-complement representation, and the mantissa is a w_m-bit binary fraction:

Number (decimal)	Number (binary)	Exponent (decimal)	Mantissa (binary)	Representation (bits)
0.5	0.1	0	.100000000000	0 000 1000 0000 0000
0.375	0.011	−1	.110000000000	0 111 1100 0000 0000
3.1415	11.001001000011 $\cdots$	2	.110010010000	0 010 1100 1001 0000
−0.1	−0.0001100110011 $\cdots$	−3	.110011001100	1 101 1100 1100 1100

The exact decimal equivalent of the last value is −0.0999755859375. Not even 0.1 can be represented exactly using fixed-precision binary fractional numbers.

Floating point representations

Floating point representations can also be investigated:

- `floatbits.c`

0.0	is	00000000 00000000 00000000 00000000
1.0	is	00111111 10000000 00000000 00000000
–1.0	is	10111111 10000000 00000000 00000000
2.0	is	01000000 00000000 00000000 00000000
10.5	is	01000001 00101000 00000000 00000000
20.1	is	01000001 10100000 11001100 11001101

More types

The (non-ANSI) extended types `long long` (64-bit integer) and `long double` (128-bit floating point value) might also come in useful at some stage.

And, to end with two programming jokes:

Why do programmers always get Christmas and Halloween confused? Because $31_{Oct} = 25_{Dec}$!.

There are 10 types of people in the world: those who know binary, and those who don't. (Boom boom).

Root finding

Given a continuous function $f()$, determine the set of values x such that $f(x) = 0$.

Why? Function often represents a point of balance between to opposing constraints (for example, gravity and frictional resistance on falling object at terminal velocity).

Also required to find maxima and minima. Turning points have zero derivative.

Three main methods:

- graphical methods;
- bracketing methods; and
- open methods.

Graphical methods

Plot the graph, and look for the roots.

Can use a graphing calculator, even if it can't identify the exact roots.

Knowledge of general shape of function, and imprecise knowledge of roots is of benefit to more principled methods.

Bisection method

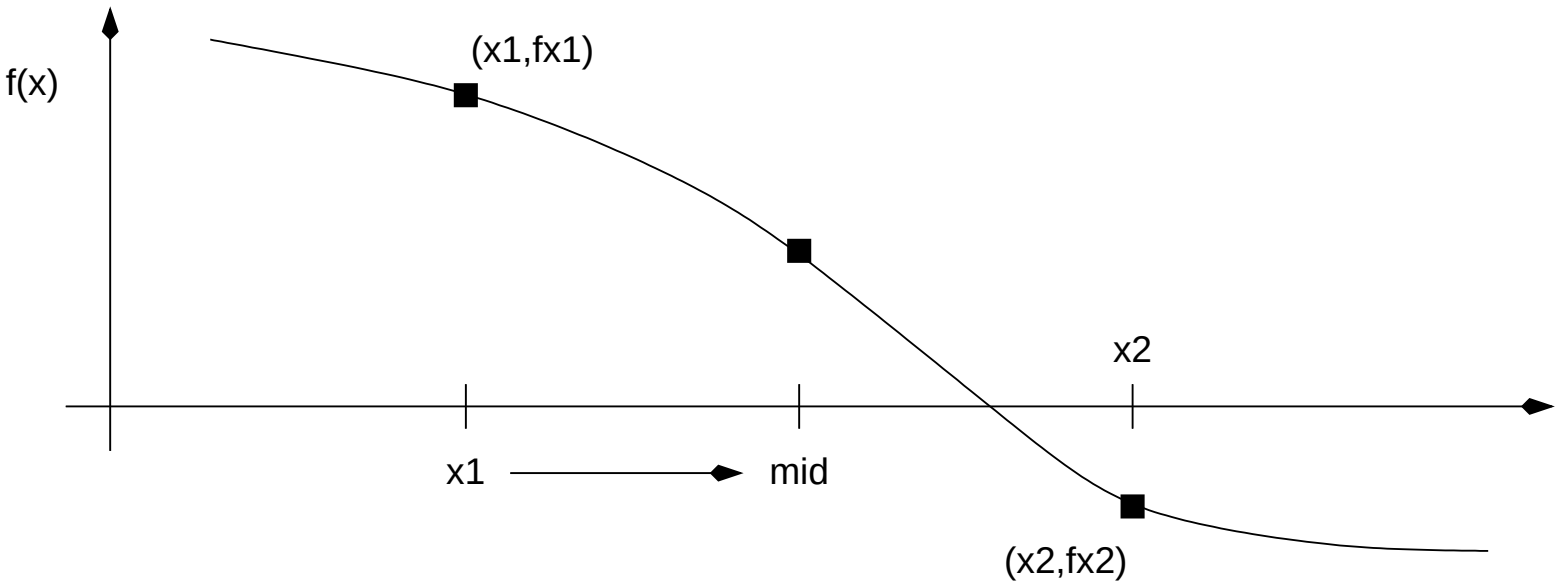
A better algorithm uses bisection to rapidly locate an approximation of the root.

Start with a pair of values x_1 and x_2 for which $f(x_1) \times f(x_2) < 0$.

At each step the mid-point $x_m = (x_1 + x_2)/2$ of the current range is tested. Keep root sandwiched between two x values at which $f(x)$ has opposite signs, by moving either x_1 or x_2 to x_m.

- `bisection.c`

Bisection method



Bisection method

The function judges termination by two different conditions:

- Whether the root has been found to the required precision (and not by a test of the form `f(x)==0.0`), and
- Whether some preset number of iterations has been exceeded.

The latter is necessary to prevent endless non-converging computation caused by a too-tight tolerance on the approximation.

In floating point arithmetic, there are no continuous functions, and every function is discrete.

Bisection example

If $f(x) = x^2 - 10$, and $x_0 = 0$ and $x_1 = 10$, the sequence of midpoint estimates x_m is

$$5, 2.5, 3.75, 3.125, 3.4375, 3.28125, 3.203125, 3.1640625$$

The correct value is $x = \sqrt{10} = 3.16227766016$.

This form of calculation can also be easily done with a spreadsheet.

Bisection convergence

Each iteration adds one bit of accuracy, or a little under one third of a decimal digit. This observation allows the number of loop iterations to be bounded.

So 32 iterations will give an accurate `float`, and 64 iterations an accurate `double`.

Bisection is robust and reliable, provided an initial interval is available. Can do this via iterative search through through x values, stepping with some suitable interval.

False position

Rather than use midpoint, could also use proportions and triangles to get a "better" next estimate,

$$x_m = \frac{f(x_2)x_1 - f(x_1)x_2}{f(x_2) - f(x_1)}.$$

This is the method of false position.

Need to be a bit more careful with termination test, best bet is to use `fabs(f(xm))<EPSILON`, because `x2-x1` might not converge.

May require fewer loop iterations than bisection to reach similar level of precision.

Convergence

In practice convergence speed is not a problem, 50 or 100 loop iterations is nothing.

But both bisection and false position are unreliable (or may fail completely) when multiple roots fall within preliminary inspection interval.

Open methods

A range of other options are also available that do not require an initial bracketing interval.

In the one-point iterative approach, the defining equation is rewritten to isolate (one instance of) x.

For example, if $f(x) = e^x - 10x + 1$, solve for x via the rewrite $x = g(x) = (e^x + 1)/10$.

Then use this function to generate a sequence:

$$x_{i+1} = g(x_i) = \frac{e^{x_i} + 1}{10}$$

Fixed point iteration

The fixed-point iteration method converges when the magnitude of the derivative of the new function $g()$ in the vicinity of the root (including the initial estimate) is less than one.

Starting at $x_0 = 1$ gives 0.37, 0.15, 0.227, 0.2256, 0.22530, 0.22527, 0.225266, 0.2252656, and so on.

But doesn't converge for $x_0 = 3.54$, which is just a little bigger than another root.

Newton-Raphson

If the slope of the function at the estimated root is known, it can be used to compute a tangent to the curve.

Starting with x_0 is an estimate of the true value x, compute a sequence of better estimates via:

$$x_{i+1} = \mathrm{NR}(x_i) = x_i - \frac{f(x_i)}{f'(x_i)} .$$

This approach requires that f is differentiable, and that the derivative is well-behaved over the interval in question. But does not require that the absolute gradient be less than one.

Newton-Raphson

The number of correct digits approximately doubles at each iteration, provided certain conditions are met.

Gives faster convergence than bisection method, but also has greater risks – when $f'(x) \approx 0$, or when $f'(x_i) = 0$ for any x_i, may have serious problems.

Newton-Raphson

Suppose that $f(x) = x^2 - 10$, and $\sqrt{10}$ is to be computed.

In this case, $f'(x) = 2x$, and hence

$$\mathrm{NR}(x) = x - \frac{x^2 - 10}{2x} = \frac{1}{2}\left(x + \frac{10}{x}\right) .$$

This yields the sequence (assuming $x_0 = 1$)

$$1.0, 5.5, 3.66, 3.196, 3.16246, 3.1622777, 3.16227766 .$$

(And the correct value is $x = \sqrt{10} = 3.162277660168379$.)

Secant method

Combining the ideas of false position and Newton-Raphson, generate a sequence from two starting points (x_0, y_0) and (x_1, y_1):

$$x_{k+1} = \frac{y_k x_{k-1} - y_{k-1} x_k}{y_k - y_{k-1}}$$

The tangent $f'(x)$ is approximated by computing the gradient between two points lying on the curve.

Multiple simultaneous equations

Systems of equations also sometimes arise:

$$\begin{aligned} f_1(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ f_3(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ &\ldots \\ f_n(x_1, x_2, x_3, \ldots, x_n) &= 0 \end{aligned}$$

where the "roots" are vectors of n values.

Multiple simultaneous equations

Example:

$$\begin{aligned} \sin x_1 - \cos x_2 &= 0 \\ x_1 + x_2 - 1 &= 0 \end{aligned}$$

has solution $(x_1, x_2) = (-1.85619, 2.85619)$.

A multidimensional version of NR can be used in a similar iterative approach, making use of partial derivatives and hyperplanes.

Root finding – summary

For simple well-behaved situations, use NR or bisection, both are fine. NR is faster, but requires that the derivative be available. Difference in speed is likely to be inconsequential.

Where there may be multiple roots: beware, both methods may have issues.

Turning points near the root being sought can give divergent outcomes with NR; and multiple roots can affect both approaches. So make sure you know the nature of the function first.

Numerical integration

Given some continuous function $f(x)$ and two limits, x_1 and x_2, compute

$$\int_{x_1}^{x_2} f(x)dx\,.$$

Useful in a range of situations where volumes of objects are required, total energy in dynamic systems, and etc.

One trapezoid

As a crude approximation, assume that $f(x)$ is linear between the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$, and compute the area of the trapezoid that is defined:

$$I = (x_2 - x_1)\frac{f(x_1) + f(x_2)}{2}.$$

But the error can be arbitrarily large. Consider $f(x) = -x^2 + k$, with $x_1 = -\sqrt{k}$ and $x_2 = \sqrt{k}$.

Trapezoidal method

To get a better estimate, break the interval up into n steps each of size $h = (x_2 - x_1)/n$. Then the area of each small trapezoid is worked out, and added to a total.

$$\begin{aligned} I &= \sum \text{trapezoids} \\ &= \sum_{i=0}^{n-1} h \cdot \frac{f(x_1 + h \cdot i) + f(x_1 + h \cdot (i+1))}{2} \\ &= \frac{h}{2}\left(f(x_1) + 2\sum_{i=1}^{n-1} f(x_1 + h \cdot i) + f(x_1 + h \cdot n) \right) . \end{aligned}$$

Trapezoidal method

The greater the value of n, the smaller the value of h, and the better the approximation.

In theory, that is.

In practice, rounding errors arising from summation of many values of comparable magnitude needs to be monitored. Techniques for progressive aggregation might be helpful.

And note that multiplication by h is factored out and only performed once, avoiding another possible source of rounding error.

Simpson's method

The trapezoidal approach fits a linear polynomial to pairs of points. If n is even (or is doubled to make it even), can fit a quadratic to consecutive triples of points.

It turns out that if x_1, x_2, and x_3 are evenly spaced, and if $P(x) = px^2 + qx + r$ is fitted through $(x_1, f(x_1))$, $(x_2, f(x_2))$, and $(x_3, f(x_3))$, then

$$\int_{x_1}^{x_3} P(x)dx = \frac{x_3 - x_1}{6} \left(f(x_1) + 4f(x_2) + f(x_3) \right) .$$

Simpson's method

Hence, Simpson's method for numerical integration:

$$I = \frac{h}{3}\left(f(x_0) + f(x_0 + h \cdot n) + 4 \sum_{i=1,3,5,\ldots}^{n-1} f(x_0 + h \cdot i) + 2 \sum_{i=2,4,6,\ldots}^{n-2} f(x_0 + h \cdot i) \right) .$$

The improved approach has significantly better convergence properties. For a given level of mathematical error it allows use of a smaller n, meaning that rounding errors are less likely to intrude.

Numeric interpolation

Suppose that instead of being given a function $f(x)$, get given some pairs of points that lie on f: $(x_1, f(x_1))$, $(x_2, f(x_2))$, $(x_3, f(x_3))$, $\cdots$ $(x_n, f(x_3))$.

And f is not known.

Then how can the roots of f, or the integral of f over a range, be estimated?

How even can single values $f(x)$ be estimated for $x \neq x_i$?

Linear interpolation

If $n = 2$, the only plausible estimate for f is as a linear combination:

$$\hat{f}_1(x) = f(x_1) + (x - x_1)\frac{f(x_2) - f(x_1)}{x_2 - x_1}.$$

The function is fitted as a straight line that passes through the two specified points.

Linear interpolation

When more that two points, an option is to construct a piecewise linear interpolation – for $x_i \leq x \leq x_{i+1}$, take

$$\hat{f}_1(x) = f(x_i) + (x - x_i)\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}.$$

That is, between each pair of consecutive (in terms of position along the x axis), assume a straight line.

Problem now is that $\hat{f}$ is continuous, but $\hat{f}'$ is not – at each point x_i, the derivative $\hat{f}'(x_i)$ may be undefined, as there may be two limits.

Newton quadratic interpolation

If wish $\hat{f}$ to be differentiable, fit a polynomial of degree $n-1$ through the n points.

Suppose three points supplied, $(x_1, y_1), (x_2, y_2), (x_3, y_3)$.

Three points define a unique quadratic. Just need to find it!

Newton quadratic interpolation

Define:

$$\hat{f}_2(x) = b_1 + b_2(x - x_1) + b_3(x - x_1)(x - x_2)$$

Considering $x = x_1$ implies $b_1 = y_1$, since $\hat{f}_2(x_1) = y_1$.

Substituting b_1, and considering $x = x_2$ implies

$$b_2 = \frac{y_2 - y_1}{x_2 - x_1},$$

since $\hat{f}_2(x_2) = y_2$.

Newton quadratic interpolation

Finally, substituting b_1 and b_2, and considering $x = x_3$ implies

$$b_3 = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1}$$

since $\hat{f}_2(x_3) = y_3$.

Newton quadratic interpolation

To get a quadratic in the usual form $\hat{f}_2(x) = px^2 + qx + r$, expand the original:

$$\hat{f}_2(x) = b_1 + b_2(x - x_1) + b_3(x - x_1)(x - x_2)$$

to determine that

$$\begin{aligned} p &= b_3 \\ q &= b_2 - b_3 x_1 - b_3 x_2 \\ r &= b_1 - b_2 x_1 + b_3 x_1 x_2 \end{aligned}$$

Example 1

Given these points:

x_i	y_i
1.00	-2.50
1.50	-1.50
2.50	3.50

Calculate a quadratic $\hat{f}_2(x) = px^2 + qx + r$ that passes through the three points.

Example 2

Consider the function

$$g(x) = \ln \frac{1}{1+x}\,.$$

Find a quadratic approximation to g that matches g at the points $x \in \{1, 2, 4\}$.

Newton polynomial interpolation

Over n points (x_i, y_i), for $1 \leq i \leq n$, define

$$F[i] = y_i \,.$$

Then from that basis recursively define

$$F[j, j-1, \ldots, i+1, i] =$$

$$\frac{F[j, j-1, \ldots, i+1] - F[j-1, \ldots, i+1, i]}{x_j - x_i} \,.$$

Finally, define $b_j = F[j, \ldots, 1]$ for $j \in \{1 \ldots n\}$.

Newton polynomial interpolation

Then, by construction, taking

$$\begin{aligned}\hat{f}_n(x) \quad = \quad & b_1 + \\ & b_2(x - x_1) + \\ & b_3(x - x_2)(x - x_1) + \\ & b_4(x - x_3)(x - x_2)(x - x_1) + \\ & \dots \\ & b_n(x - x_{n-1})(x - x_{n-2}) \cdots (x - x_2)(x - x_1)\end{aligned}$$

yields the required polynomial.

Splines

The Newton polynomial is continuous and differentiable, but the function is likely to diverge rapidly outside the zone for which points are supplied, and may also overshoot/return if there are step changes within the spanned zone.

In a spline, component polynomials are pieced together, with a section over the interval $[x_i, x_{i+1}]$ connecting with similar sections left and right.

The simplest spline is the piecewise linear form.

Splines

If the degree of the component polynomials is increased (from first degree linear), added freedom is added to the solution space (compared to linear).

Additional requirements can then be added without over-constraining the system of equations:

- $\hat{f}'(x_i)$ can be required to be continuous; and
- $\hat{f}''(x_i)$ can be required to be continuous.

The first is achieved in a quadratic spline, the second added in a cubic spline.

Splines

A cubic spline is continuous at each knot point, and its first and second derivatives are also continuous. The two "spare" coefficients are usually set by taking $\hat{f}''(x_1) = \hat{f}''(x_2) = 0$.

The maths gets a (little) bit harder, and there are more coefficients to be stored, plus (slightly) more complex evaluation.

But a cubic spline is normally visually attractive, gives smooth transitions of gradient, and has smaller errors on well-behaved functions.