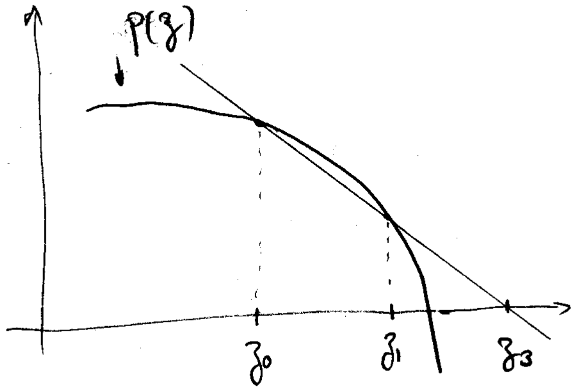
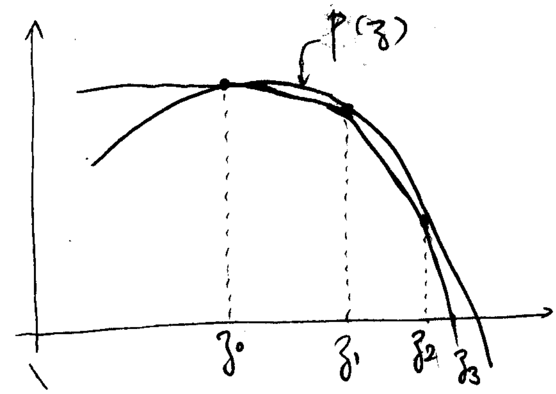
Another alternative is *Müller's method* which works with same idea as secant method but using a *quadratic model* of the function (which requires 3 points as opposed to 2 for linear model in secant method).



SECANT METHOD

function $p(z)$ is approx. by a *linear model*: secant passing through points $(z_0, p(z_0))$ $(z_1, p(z_1))$.
next iterate is *root of linear model*



MÜLLER'S METHOD

function $p(z)$ is approx. by a *quadratic model*: parabola passing through points $(z_0, p(z_0))$, $(z_1, p(z_1))$ and $(z_2, p(z_2))$.
next iterate is *root of quadratic*

In pseudocode:

Muller's method

Given z_0, z_1, z_2
for $k = 1 \ldots$ maxit

- find coeff a, b, c of quadratic $q(z) = a(z - z_2)^2 + b(z - z_2) + c$ $(z_0, p(z_0)), (z_1, p(z_1)), (z_2, p(z_2))$ — (2.18) – (2.20) See book. We shall see an easy way of determining a, b, c when we get to interp
- z_3 = root of $q(z)$ closest to z_2
- if $|z_3 - z_2| < tol$ stop.
- $z_0 = z_1$; $z_1 = z_2$; $z_2 = z_3$; prep for next iteration.

Advantage of Müller's method is that:

- if used close to real root it should not use complex arithmetic
- if used close to complex root then complex numbers occur naturally since roots of $q(z)$ may be complex

One word of caution on finding roots of quadratic:

$$q(z) = az^2 + bz + c$$

Usual formula:

$$z_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

may not be stable numerically because of catastrophic cancellation if b is large and ac small

in fact: if $b > 0$ then $z_1 = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$ could be unstable (subtracting 2 numbers of similar magnitude)

$z_2 = \dfrac{-b - \sqrt{b^2 - 4ac}}{2a}$ is fine. (adding 2 numbers of same magnitude)

Fortunately we can find alternate formulas where this catastrophic cancellation does not occur: (say $b > 0$ for simplicity)

$$z_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \; \frac{+b + \sqrt{b^2 - 4ac}}{+b + \sqrt{b^2 - 4ac}}$$

$$= \frac{b^2 - 4ac - b^2}{(b + \sqrt{b^2 - 4ac})2a} = -\frac{2c}{b + \sqrt{b^2 - 4ac}}$$

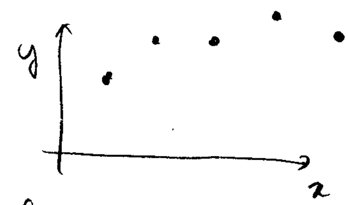
← no catastrophic cancellation anymore!

The choice of root in Müller's method is:

$$z_3 - z_2 = \frac{-2c}{b + \operatorname{sgn}(b)\sqrt{b^2 - 4ac}}$$

no catastrophic cancellation
and denominator is largest in magnitude
$\Rightarrow$ <u>z_3 is closest root to z_2</u>.

Idea:



interpolation: given some data points find a polynomial that passes through all data points.

extremely useful to:

- fill-in gaps in data from e.g. measurements
- numerical integration: integrating interpolating polys is easy!
- numerical differentiation: differentiating interpolating poly is easy!

Why would we expect to have a good match between function and polynomial?

Theorem Weierstrass Let $f \in C[a,b]$.

$$\forall \varepsilon > 0 \quad \exists \text{ poly. } p(x) \text{ s.t. } \forall x \quad |f(x) - p(x)| < \varepsilon$$

i.e. polynomials uniformly approx continuous functions.

Remark: Taylor's polynomial does approximate "well" a function, but approximation is local meaning it quickly degrades far away from point x_0 where we do the expansion.

Interpolation allows us to get poly that is close to function over some interval.

§ 3.1 Polynomial interpolation

Theorem on polynomial interpolation: If $x_0, x_1, \dots, x_n$ are $n+1$ distinct real numbers, then for arbitrary values $y_0, y_1, \dots, y_n$ there is a unique polynomial p_n of degree at most n for which:

$$p_n(x_i) = y_i, \quad i = 0, \dots, n.$$

proof: i) uniqueness: assume there are two interpolating poly p_n and q_n of degree at most n s.t.

$$p_n(x_i) = y_i = q_n(y_i), \quad i = 0, \dots, n.$$

then $r_n(z) = p_n(z) - q_n(z)$ is a poly of degree $\leq n$ with $n+1 > n$ distinct roots $\Rightarrow r_n(z) \equiv 0$ (FTA)

$$\Rightarrow p_n(z) = q_n(z).$$

ii) For existence part of theorem we work by induction.

clearly:

when $n = 0$ there is $p_0(z) \equiv$ constant s.t. $x_0 = y_0$.

now assume we have a polynomial p_k of degree k s.t. $p_k(x_i) = y_i \quad i = 0, \dots, k$.

Let p_{k+1} be of the form:

$$p_{k+1}(x) = p_k(x) + c(x - x_0)(x - x_1)\cdots(x - x_k)$$

then clearly:

$$p_{k+1}(x_i) = p_k(x_i) \quad \text{for } i = 0, \dots, k$$

we determine constant c from:

$$p_{k+1}(x_{k+1}) = p_k(x_{k+1}) + c(x_{k+1} - x_0)(x_{k+1} - x_1)\cdots(x_{k+1} - x_k)$$

$$c = \frac{p_{k+1}(x_{k+1}) - p_k(x_{k+1})}{(x_{k+1} - x_0)(x_{k+1} - x_1)\cdots(x_{k+1} - x_k)}$$

note the denominator is non-zero because we assumed all the nodes to be distinct.

Although interpolating polynomial is unique there are several practical way of writing it. The first one is:

Newton form of interp. poly

Unravelling the polynomial that is constructed in previous proof we get:

$$p_k(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + \dots + c_k(x-x_0)\cdots(x-x_{k-1})$$

$$= \sum_{i=0}^{k} c_i \prod_{j=0}^{i-1} (x-x_j)$$

(w/ convention that $\prod_{j=0}^{-1} (x-x_j) = 1$)

Evaluation of Newton's form of interp poly

This can be done using Horner's algorithm or nested multiplication:

Idea: Imagine we would like to evaluate an expression of the kind:

$$u = \sum_{i=0}^{k} c_i \prod_{j=0}^{i-1} d_j = c_0 + c_1 d_0 + c_2 d_0 d_1 + c_3 d_0 d_1 d_2 + \dots + c_k d_0 d_1 \dots d_{k-1}$$

$$= \Big(\underbrace{\big(\overbrace{(\overbrace{(c_k)}^{u_k} d_{k-1} + c_{k-1})}^{u_{k-1}} d_{k-2} + c_{k-2}\big)}_{u_{k-2}} d_{k-3} + \dots + c_1\Big) d_0 + c_0$$

To compute u we can organize calculations as follows.

$$u_k = c_k$$
$$u_{k-1} = u_k d_{k-1} + c_{k-1}$$
$$u_{k-2} = u_{k-1} d_{k-2} + c_{k-2}$$
$$\vdots$$
$$u_0 = u_1 d_0 + c_0$$

Algorithm:

$u = c_k$

for $i = k-1 : -1 : 0$

  $u = u d_i + c_i$

Or going back to Newton's form; if we would like to evaluate $u = p_k(t)$:

$$\begin{array}{l} u = c_k \\ \text{for } i = k-1 : -1 : 0 \\ \quad u = (t - x_i)u + c_i \end{array}$$

Now how do we evaluate the coefficients c_k? From proof we get:

$$c_k = \frac{y_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})}$$

which we can translate into the following algorithm:

$$\begin{array}{l} c_0 = y_0 \\ \text{for } k = 1..n \text{ do} \\ \quad d = x_k - x_{k-1} \\ \quad u = c_{k-1} \\ \quad \left.\begin{array}{l} \text{for } i = k-2 : -1 : 0 \\ \quad u = u(x_k - x_i) + c_i \\ \quad d = d(x_k - x_i) \end{array}\right\} \text{evaluates } u = p_{k-1}(x_k),\ d = \prod_{i=0}^{k-1}(x_k - x_i) \\ \quad c_k = \dfrac{y_k - u}{d}. \end{array}$$

<u>Note</u>: this is not a method that is used in practice, it is just a straightforward translation of the proof... More efficient algorithms are <u>divided differences</u> and <u>Neville's algo</u>.

Lagrange form of interpolation polynomial

Here is another form of the interp. poly that is nice for the theory: Again recall we are given $n+1$ points (x_i, y_i) $i=0,\dots,n$, where the x_i are distinct.

$$P_n(x) = y_0 l_0(x) + y_1 l_1(x) + \dots + y_n l_n(x) = \sum_{k=0}^{n} y_i l_k(x).$$

here we want $l_i(x)$ to be polynomials depending on abscissas x_i but <u>not</u> on ordinates y_i. The interpolation condition $\Rightarrow$

$$p_n(x_i) = y_i = \sum_{k=0}^{n} y_k l_k(x_i) \quad \text{for all } i$$

$\rightsquigarrow$ works if we require $\boxed{l_k(x_i) = \delta_{ki}} = \begin{cases} 1 & \text{if } k=i \\ 0 & \text{otherwise} \end{cases}$

(Kronecker δ)

How can we get such polynomials?

Take for example $l_0(x)$:

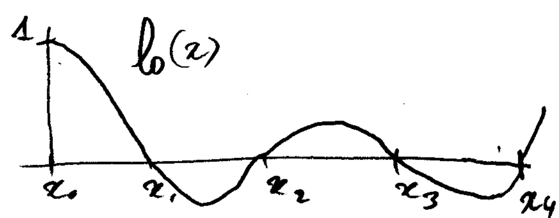
since $l_0(x_i) = 0$ for $i = 1,\dots,n$:

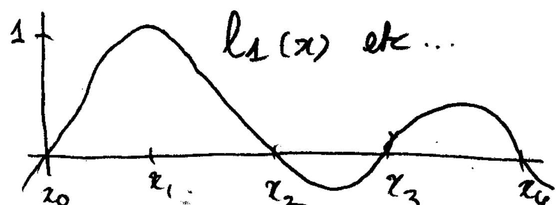
$l_0(x) = c(x-x_1)(x-x_2)\cdots(x-x_n)$

since $l_0(x_0) = 1$ we have:

$1 = l_0(x_0) = c\,(x_0-x_1)(x_0-x_2)\dots(x_0-x_n)$

$$\Rightarrow \quad l_0(x) = \frac{(x-x_1)(x-x_2)\dots(x-x_n)}{(x_0-x_1)(x_0-x_2)\dots(x_0-x_n)}$$





In general:

$$\boxed{l_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-x_j}{x_i-x_j}}$$

⚠ denominator is $\neq 0$ because nodes are different

"cardinal polynomials"

Here is yet another (bad!) way of getting interpolation poly:

imagine we wanted the interp poly of the form:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$$

interpolation conditions:

$$p(x_i) = y_i \quad , i = 0, \dots, n \qquad (n+1 \text{ eq}, n+1 \text{ unknowns})$$

can be written in linear system form:

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 \dots & x_0^n \\ 1 & x_1 & x_1^2 & x_1^3 \dots & x_1^n \\ & & \vdots & & \\ 1 & x_n & x_n^2 & x_n^3 \dots & x_n^n \end{bmatrix}} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}.$$

Vandermonde matrix: by interpolation theorem should be invertible if x_i are distinct.

however matrix is ill-conditioned. We shall see in chap 6 what this means exactly, but intuively it means very small errors on the y_i can lead to huge changes in the a_i.

$\rightarrow$ not a stable way of computing interp poly!

If we did the same with Lagrange interp polyn.

$$\underbrace{\begin{bmatrix} l_0(x_0) & l_0(x_1) \dots & l_0(x_n) \\ l_1(x_0) & l_1(x_1) \dots & l_1(x_n) \\ \vdots & & \\ l_n(x_0) & l_n(x_1) \dots & l_n(x_n) \end{bmatrix}}_{\text{identity matrix}} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$\rightsquigarrow$ very stable.

Lagrange interp: • good for theory

• good if one has to interpolate many data sets (x_i, y_i) $i=0,\dots n$, with x_i the same among data sets (compute $l_i(x)$ once and reuse)

Newton form: • efficient and easy to evaluate.

• accomodates adding new nodes easily (coeff $c_0, \dots, c_k$ depend on nodes $x_0 \dots x_k$)

Interpolation error (very similar to Taylor's theorem)

Theorem on polynomial interp error

Let $f \in C^{n+1}[a,b]$ and let p be the poly of degree $\leq n$ that interpolates f at $n+1$ distinct points $x_0, x_1, \dots, x_n$ in $[a,b]$. Then $\forall x \in [a,b]$, $\exists \xi_x \in (a,b)$ s.t.

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{i=0}^{n} (x - x_i).$$

Proof: • case $x = x_i$: theorem is trivial to prove $0=0$.

• case $x \neq x_i$: (x fixed)

Let $w(t) = \prod_{i=0}^{n} (t - x_i)$ $\quad \phi \equiv f - p - \lambda w$

where $\lambda \in \mathbb{R}$ is chosen s.t. $\phi(x) = 0$ i.e.

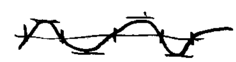
$$\lambda = \frac{f(x) - p(x)}{w(x)}$$

⚠ denominator is nonzero because ... ?

$\phi \in C^{n+1}[a,b]$ and ϕ is zero at points: $x_0, x_1, \dots, x_n, x$

$\Rightarrow$ ϕ has at least $n+2$ zeros in $[a,b]$

ϕ' ——— $n+1$ ——— by Rolles theorem



ϕ'' ——— n ———

$\vdots$

$\phi^{(n+1)}$ ——— 1 ———

call this zero ξ_x:

$$\phi^{(n+1)}(\xi_x) = 0 = f^{(n+1)}(\xi_x) - (n+1)!\overbrace{\left(\frac{f(x)-p(x)}{w(x)}\right)}^{\lambda}$$

This is the proof!

$$f(x)-p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x)\, w(x)$$

$$= \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{i=0}^{n} (x-x_i)$$

§3.2 Divided differences

Let f be a function which is known at $n+1$ distinct nodes $x_0, x_1, \dots x_n$. We know there is a unique polynomial $P(x)$ of degree $\leq n$ interpolating f at the nodes x_i, ie.

$$p(x_i) = f(x_i), \quad i = 0, 1, \dots, n.$$

Recall that interp. poly. can be written in Newton form:

$$p(x) = \sum_{j=0}^{n} c_j q_j(x)$$

where

$$q_0(x) = 1$$
$$q_1(x) = (x - x_0)$$
$$q_2(x) = (x-x_0)(x-x_1).$$
$$\vdots$$
$$q_n(x) = (x-x_0)(x-x_1)\dots(x-x_n)$$

Also recall that

C_0 is found s.t. $\sum_{j=0}^{0} c_j q_j(x)$ interp. f at nodes x_0.

C_1 ———— $\sum_{j=0}^{1} c_j q_j(x)$ ———— x_0, x_1

$\vdots$

C_k ———— $\sum_{j=0}^{k} c_j q_j(x)$ ———— $x_0, x_1, \dots, x_k$.

$\rightsquigarrow$ in general c_k depends only on nodes $x_0, \dots, x_k$ and $f(x_0), \dots, f(x_k)$.

Because of this dependency we denote:

$$C_n = f[x_0, x_1, x_2 \dots, x_n]$$

In other words

$$f[x_0, x_1, \dots, x_n] \equiv \text{coeff of } q_n \text{ in interp poly}$$

But $\quad q_n(x) = (x-x_0)(x-x_1)\cdots(x-x_{n-1}) = x^n + \text{lower order terms}$

$\rightsquigarrow \quad f[x_0, x_1, \dots, x_n] = $ coeff in front of x^n in poly interp f at points $x_0, x_1, \dots, x_n$.

$f[x_0, \dots, x_n]$ is called a <u>divided difference</u> of f. The reason comes from figuring out the first few divided differences:

$$\boxed{f[x_0] = f(x_0)}$$ (since $f[x_0] =$ coeff in front of const interpolating f at x_0)

Now the polynomial interpolating f at x_0, x_1 is:

$$p(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

$$\Rightarrow \boxed{f[x_0\, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}}$$ = divided difference nomenclature origin.

In general:

$$(*) \quad p(x) = \sum_{k=0}^{n} c_k q_k(x) = \sum_{k=0}^{n} f[x_0, x_1, \dots x_k] \prod_{j=0}^{k-1} (x - x_j)$$

is the interpolating poly of f at $x_0, \dots, x_n$

<u>Note</u>: if we would like to interpolate only on $x_0, \dots, x_m$, with $m \leq n$, the interp poly will be the sum $(*)$ truncated up to $k = m$.

Why go through this trouble? Because of the following nice relationship we can evaluate divided differences in an efficient manner:

Theorem (Higher order divided differences)

$$f[x_0,\dots,x_n] = \frac{f[x_1,x_2,\dots,x_n] - f[x_0,x_1,\dots,x_{n-1}]}{x_n - x_0}$$

proof: let $p_k \equiv$ poly of degree $\le k$ interpolating f at $x_0, x_1, \dots, x_k$.

$q \equiv$ poly of degree $\le n-1$ interp. f at $x_1, x_2, \dots, x_n$

then:
$$p_n(x) = q(x) + \frac{x - x_n}{x_n - x_0}\left[q(x) - p_{n-1}(x)\right] \quad (*)$$

why? First both poly have degree $\le n$ and:

for $i = 1, \dots, n-1$, $\underbrace{q(x_i)}_{f(x_i)} + \frac{x_i - x_n}{x_n - x_0}\left[\underbrace{q(x_i)}_{f(x_i)} - \underbrace{p_{n-1}(x_i)}_{f(x_i)}\right] = f(x_i) = p_n(x_i)$

also:
$$q(x_0) + \overbrace{\frac{x_0 - x_n}{x_n - x_0}}^{-1}\left[q(x_0) - p_{n-1}(x_0)\right] = p_{n-1}(x_0) = p_n(x_0) = f(x_0)$$

$$q(x_n) + 0\,[\cdots] = f(x_n) = p_n(x_n)$$

thus the poly on both sides of $(*)$ match at $n+1$ points $\Rightarrow$ they must be equal!

By matching coeff of degree n we get:

$$p_n(x) = x^n f[x_0, \dots x_n] + \text{l.o.t}$$

$$= \text{l.o.t.} + \frac{x^n}{x_n - x_0}\left[f[x_1 x_2 \cdots x_n] - f[x_0, x_1, \dots, x_n] \right]$$

QED.

Since we are free to choose the interpolation points, we get:

$$f[x_i, x_{i+1} \cdots, x_{i+j}] = \frac{f[x_{i+1} x_{i+2}, \dots, x_{i+j}] - f[x_i, x_{i+1}, \dots, x_{i+j-1}]}{x_{i+j} - x_i}.$$

$f[x_i] \equiv$ div. diff of order 0

$f[x_i, x_{i+1}] \equiv$ " " " " 1

$f[x_i, x_{i+1}, x_{i+2}] \equiv$ " " " " 2

etc...

The computation of divided differences is usually organized as a table:

x_0	$f[x_0]$	$f[x_0, x_1]$	$f[x_0, x_1 x_2]$	$f[x_0, x_1, x_2 x_3]$
x_1	$f[x_1]$	$f[x_1 x_2]$	$f[x_1, x_2, x_3]$	
x_2	$f[x_2]$	$f[x_2, x_3]$		
x_3	$f[x_3]$			
given data		1st order	2nd order	3rd order

Each column corresponds to div diff of the same order and the entries in columns after the dividing line can be computed from entries of preceding column by applying divided differences theorem.