April 7, 2020

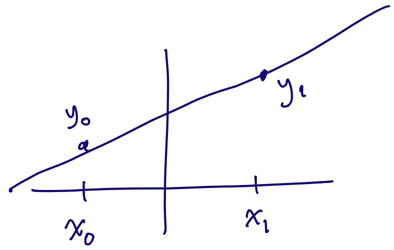
Numerical Analysis

Last Time: - Finished Jacobi's Algorithm for computing eigenvalues/vectors for a real symmetric matrix

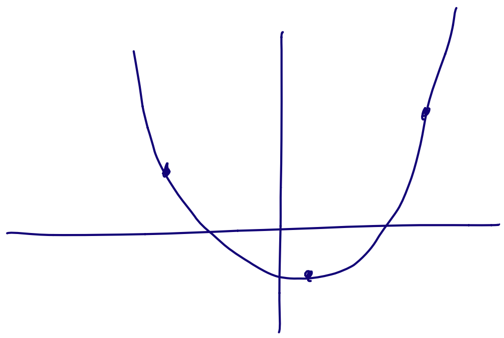
Next up: Polynomial Interpolation

Since computers can only multiply/add, basically the only functions that your computer can evaluate are polynomials.

Ex: • 2 points in the xy-plane define a line.



• 3 points define a parabola (a deg 2 polynomial)



In general, $n+1$ unique points in the xy-plane define a polynomial of degree n:

$$p(x) = \underbrace{a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0}$$

$$(*) \quad \left.\begin{aligned} p(x_0) &= y_0 \\ p(x_1) &= y_1 \\ &\vdots \\ p(x_n) &= y_n \end{aligned}\right\} \text{ $n+1$ equations for the $n+1$ unknowns a_j in}$$

The point of Interpolation

$\boxed{\text{Function Approximation}}$ Most functions (i.e., $\cos x$, solutions to differential equations, etc.) are <u>not</u> polynomials, and do <u>not</u> have closed form solutions. <u>However</u>, most of the time they can be <u>locally</u> approximated via polynomials (just think Taylor series).

$\Rightarrow$ Polynomial interpolation is at the core of <u>Numerical Analysis</u>.

With this in mind, given (x_j, y_j)'s, $j = 0, .., n$ with $x_j \in [a,b]$, how do we compute the <u>interpolant</u>:

<u>Option 1</u> Solve for the coefficients in $p_n(x) = a_0 + a_1 x + \ldots + a_n x^n$.

(*) Can be written in matrix form as:

$$\underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ & & \vdots & & \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}}_{\text{Vandermonde Matrix } = A} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

Unless the x_j's are chosen very carefully, the relative condition number K grows exponentially.

So: If the x_j's are distinct, A is <u>formally</u> invertible, but horribly ill-conditioned. Never try to numerically invert it.

Option 2 The coefficients a_j usually don't matter — the goal is usually to evaluate p_n at some new point $x = x_j$.

While the polynomial p_n is unique, there are many ways to construct/evaluate it, the most common of which is the Lagrange Interpolating Polynomial.

particular polynomial   subspace of polynomials of degree $\leq n$

Idea: Construct a sequence of polynomials $L_k \in P_n$ such that:

$$L_k(x_j) = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases}$$

If this is possible, then $p_n(x) = \sum_{k=0}^{n} y_k L_k(x)$ is the interpolating polynomial for the data (x_j, y_j), $j = 0 \dots n$.

The construction of such polynomials L_k is straightforward:

$$L_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j} \qquad (*)$$

$$= \left(\prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{1}{(x_k - x_j)} \right) \left(\prod_{\substack{j=0 \\ j \neq k}}^{n} (x - x_j) \right)$$

Theorem Given data (x_j, y_j) $j = 0 \dots n$, there exist a unique polynomial $p_n \in P_n$ such that $p_n(x_j) = y_j$.

Proof Existence: Immediately follows from the Lagrange formula $(*)$

Uniqueness: See textbook.

The form of the interpolating polynomial $p_n(x) = \sum L_k(x) y_k$ is referred to as the "Lagrange interpolation formula of degree n".

There are a few questions that can be asked about p_n at this point:

$\boxed{Q1}$ If the points (x_j, y_j) come from a smooth function, f what is the error between p_n and the function f:

$\boxed{Q2}$ What is the cost of evaluating p_n? If a new data point is added, (x_{n+1}, y_{n+1}), what is the cost of updating p_n?

$\boxed{Q3}$ In floating point arithmetic, is the evaluation of p_n stable?

$\boxed{Q1}$ Note, if $y_j = f(x_j)$, then $p_n(x_j) = y_j = f(x_j)$ by construction. If $x \neq x_j$, then

Theorem: Let $f \in C^{n+1}[a,b]$. For $x \in [a,b]$, there exists a $\xi = \xi(x) \in (a,b)$ such that

$$f(x) - p_n(x) = \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}}_{\text{Similar to Taylor's thm}} \underbrace{\prod_{j=0}^{n} (x - x_j)}_{\text{Depends highly on the choice of interpolation points.}}$$

Exact pointwise error.

Moreover:

$$\left| f(x) - p_n(x) \right| \le \frac{M_{n+1}}{(n+1)!} \left| \Pi_{n+1}(x) \right|$$

where $\displaystyle M_{n+1} = \max_{t \in [a,b]} \left| f^{(n+1)}(t) \right|$

$$\Pi_{n+1}(x) = \prod_{j=0}^{n} (x - x_j)$$

Proof is detailed, will not go through it, see text.

Two takeaway points:

1. Only useful if M_{n+1} can be computed.
2. The interpolation error highly depends on <u>where the nodes x_j are located.</u>

This will be very important later on.

$\boxed{Q2}$ The cost of evaluating p_n depends on the form it is written in.

<u>Lagrange Form</u>: $\displaystyle p_n(x) = \underbrace{\sum_{k=0}^{n} y_k L_k(x)}_{\substack{n+1 \text{ mult} \\ n \text{ adds}}}$, $\displaystyle L_k(x) = \underbrace{\prod_{\substack{j=0 \\ j \ne k}}^{n} \underbrace{\frac{x - x_j}{x_k - x_j}}_{3 \text{ flops}}}_{3(n-1) \text{ flops.}}$

$\Rightarrow (n+1)\,3(n-1)$ flops to evaluate all L_k's.

$\Rightarrow$ overall, $O(n^2)$ flops to evaluate p_n in Lagrange form.

Compare this with Horner's Method:

If the coefficients $a_0, \dots, a_n$ are known in

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$$ the we can

rewrite p_n as:

$$p_n(x) = a_0 + x\left(a_1 + a_2 x + \dots + a_n x^{n-1}\right)$$

$$= a_0 + x\left(a_1 + x(a_2 + a_3 x + \dots + a_n x^{n-2}\right)$$

$$= a_0 + x(a_1 + x(a_2 + x(\underbrace{\cdots}_{b_{n-1} = a_{n-1} + a_n x})$$

$$b_{n-2} = a_{n-2} + b_{n-1} x$$

$b_{n-1} = a_{n-1} + a_n x$ (1 mult, 1 add)

$b_{n-2} = a_{n-2} + b_{n-1} x$ (1 mult, 1 add)

$\vdots$

$b_0 = a_0 + b_1 x$ (1 mult, 1 add)

$= p_n(x) \Rightarrow \boxed{2n \text{ flops}}$

This means that the Lagrange Form is very inefficient.

Is there a better form?

$\boxed{Q3}$ The numerical stability of evaluating p_n in Lagrange form:

Short story: The basic Lagrange form $p_n(x) = \sum_{k=0}^{n} y_k L_k(x)$ can be unstable (i.e. have large condition number).

(Ex: overflow/underflow, roundoff error, etc.)

Alternative form next class.