Polynomial Interpolation

A. Poner, Spring 2021

The next few weeks will focus on function approximation: How to represent, evaluate, and operate (differentiate, integrate) nonlinear functions on a computer.

This may appear easy for built-in functions like sin / cos / exp / ln for which we can also analytically compute derivatives & integrals but this is misleading.

After all, how does the computer evaluate exp(x) after all?
And what about more complicated or non-standard functions?

As we know from Taylor series, any smooth function can be locally approximated by a polynomial, such as the Taylor series. But Taylor series requires choosing a specific point around which we expand.
What if we want to approximate a function $f(x)$ over an interval $x \in [a,b]$ by a polynomial $p(x)$?

Why? Polynomials are easy to evaluate (only multiplication and addition, not even division), easy to integrate, differentiate, etc.
Furthermore, a fundamental theorem in analysis tells us that:

Weierstrass Approximation theorem

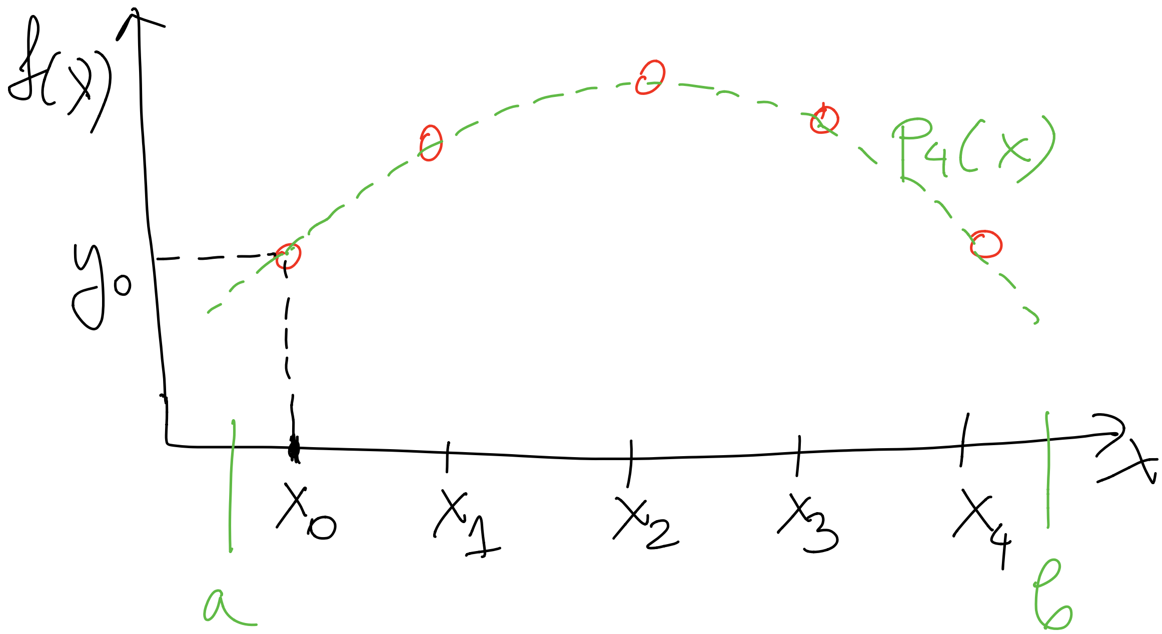
$\forall \varepsilon > 0$, $\exists p(x)$ s.t.

$$\max |f(x) - p(x)| < \varepsilon$$

But we don't know what degree $p(x)$ is, so this doesn't help us numerically.
We want to restrict the degree of the polynomial

Interpolation

One way to go about constructing an actual polynomial $P_n(x) \simeq f(x)$ on $[a,b]$ of finite degree n is to choose a set of $n+1$ nodes and evaluate f at nodes:



and find the interpolating polynomial – polynomial that passes through the points.

$\boxed{\text{(n+1) points uniquely define a polynomial of degree } n}$

How to find it?

$$P_n = a_n x^n + a_{n-1} x^{n-1} \dots + a_1 x + a_0$$

(n+1) unknown coefficients

$$\vec{a} = [a_0, a_1, \dots, a_n]$$

$$\left\{ \begin{array}{l} P(x_0) = y_0 \\ P(x_1) = y_1 \\ \vdots \\ P(x_n) = y_n \end{array} \right\} \text{(n+1) linear equations for } \vec{a}$$

$$\sum_{k=0}^{n} a_k x_i^k = y_i \,, \quad i = 0, \dots, n+1$$

5

In matrix form

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & & \cdots & & \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}}_{\text{Vandermonde matrix } V} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

$$\overleftrightarrow{V}\,\vec{a} = \vec{y}$$

is a linear system we can solve using LU factorization.

<u>theorem</u>: V is invertible if nodes are distinct, because

$$\det(V) = \prod_{j<k} (x_k - x_j) \neq 0$$

Aside: Proof that $p_n(x)$ is unique. Assume there was another polynomial $q_n(x)$ such that

$$q_n(x_i) = y_i$$

$$p_n(x_i) = y_i$$

$\Rightarrow$ $r_n = p_n - q_n$ (= polynomial of degree n) has $n+1$ zeros $x_0, \dots, x_n$ $\Rightarrow$

$$r_n \sim (x-x_0)(x-x_1)\dots(x-x_n)$$

$=$ polynomial of degree $n+1$ which is a contradiction

6 1/2

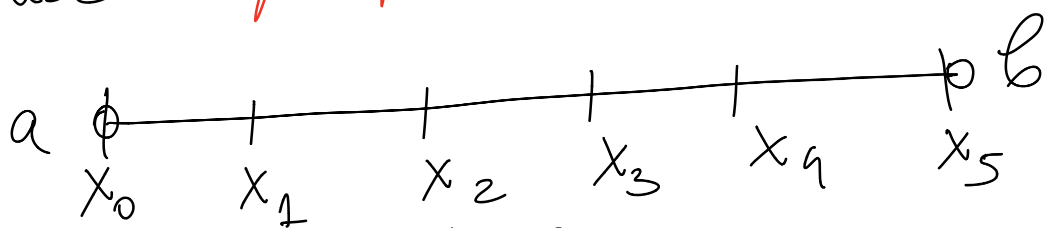
Note that we encountered this matrix before already when we talked about <u>fitting</u> a polynomial through data — there the degree n was (much) smaller than the number of points so A was not square.

This is why in MATLAB the same function polyfit does both polynomial interpolation and fitting.

<u>Observation</u>: The Vandermonde matrix is generally very ill-conditioned for large n

(unless nodes are chosen carefully)
We should expect some problems!

Most obvious choice is to use equispaced nodes



h - grid spacing

$$\begin{cases} x_i = i \cdot h + a \\ h = \dfrac{b-a}{n} \end{cases}$$

(We will learn that this is NOT a good choice...)

Since V is ill-conditioned, and solving $Va = y$ costs $O(n^3)$ FLOPS, we should look for another way!

Often we don't care about the coefficients a_k, we just want to be able to evaluate the polynomial (efficiently) at a new point x.

There are <u>many</u> ways to do this, so let's discuss a few.

The right way to think about this is via <u>abstract linear algebra</u>

Namely, polynomials of degree n form a linear space $\mathcal{P}_n$ of dimension $(n+1)$.

The standard basis for this space are the <u>monomials</u> x^k

But is there a better basis for polynomial interpolation.

The best basis would be one for which

$$\overleftrightarrow{V} \rightarrow \text{identity matrix}$$

$$\Rightarrow \vec{a} = \vec{y}$$

Can we find this basis?

(Drop subscript n for brevity)

$$P(x) = \sum_{k=0}^{n} a_k L_k(x)$$

where $\{L_k\}$ is the new basis comprised of polynomials of degree n. $\vec{p} = \overleftrightarrow{V} \vec{a}$

$$P(x_i) = \sum a_k L_k(x_i)$$

$$= V_{ik} a_k$$

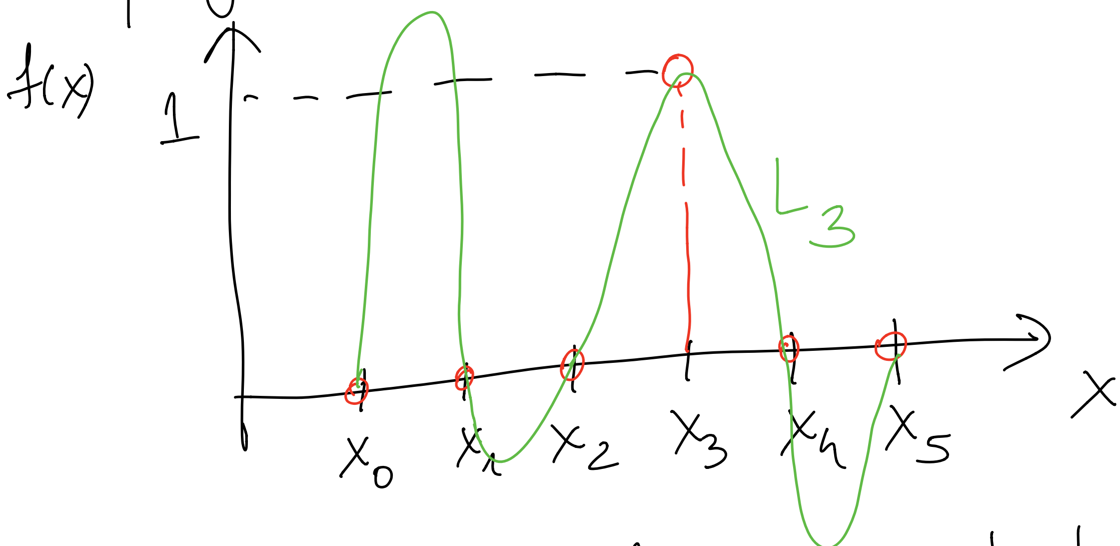
$$\Rightarrow \quad V_{ik} = L_k(x_i) = \delta_{ik}$$

$$\delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

$\uparrow$

Kronecker symbol

So L_k is the interpolating polynomial of :



We can in fact immediately write a formula for L_k since we know its roots!

$$L_k(x) = c \prod_{\substack{j=0 \\ j \neq k}}^{n} (x - x_j)$$

$$L_k(x_k) = 1 = c \prod_{j \neq k} (x_k - x_j)$$

$$\Rightarrow \quad c = \frac{1}{\prod_{j \neq k} (x_k - x_j)}$$

$$L_k(x) = \left(\frac{1}{\prod_{i \neq k} (x_k - x_i)} \right) \prod_{j \neq k} (x - x_j)$$

$$\boxed{\begin{aligned} P_n(x) &= \sum y_k L_k(x) \\ &\simeq f(x) \end{aligned}}$$

Lagrange formula for interpolating polynomial.

OK, great, now we have a way to obtain & evaluate $p(x)$ without solving ill-conditioned systems.

It doesn't mean we are done, however. ~~As~~ good numerical analysits we have to ask:

1. How expensive is it to evaluate $p_n(x)$?
   If we add a new node, can we speed things up by re-using some prior computations?

2. Can we evaluate $p_n(x)$ accurately in floating point arithmetic (i.e., with 16 digits)?

3. <u>Most important</u>: How good of an approximation of $f(x)$ is $p(x)$ for n nodes?

Question #1: Efficiency

If we are given $p_n(x)$ in the monomial basis, we can evaluate it super efficiently using Horner's method:

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

$$= a_0 + x\Big(\underbrace{a_1 + a_2 x + \dots + a_n x^{n-1}}_{p_{n-1}(x)}\Big)$$

$$= a_0 + x\Big(a_1 + x\big(\underbrace{a_2 + a_3 x + \dots + a_n x^{n-2}}_{p_{n-2}}\big)\Big)$$

Giving:

$$\begin{cases} b_{n-1} = a_{n-1} + a_n x \\ b_{n-2} = a_{n-2} + b_{n-1} x \\ \vdots \\ b_0 = a_0 + b_1 x = p(x) \end{cases}$$

15

Horner's scheme requires only n multiplications & n additions

$\Rightarrow O(n)$ FLOPS

(cannot get faster than that)

By contrast, Lagrange's form costs $O(n^2)$ FLOPs to evaluate:

$$L_k(x) = \prod_{j \neq k} \frac{x - x_j}{x_k - x_j} = 3(n-1) \text{ FLOPS}$$

(1 division, 2 subtract)

for each $k = 0, \dots, n+1$

$\Rightarrow (n+1)\,3(n-1) = O(n^2)$

FLOPS to evaluate each Lagrange polynomial — not optimal but better than $O(n^3)$

Question # 2 : Stability

It turns out Lagrange formula can suffer from numerical roundoff (floating-point) error and we can loose digits.

So we need an alternative.

The fix is not very intuitive:

$$\underbrace{P_n(x) = \sum_{k=0}^{n} \left(\prod_{j \neq k} \frac{x - x_j}{x_k - x_j} \right) y_k}_{\text{Lagrange formula}}$$

$$= \sum_k \underbrace{\left(\prod_j (x - x_j) \right)}_{\text{does not depend on } k} \frac{1}{x - x_k} \left(\prod_{j \neq k} \frac{1}{x_k - x_j} \right) y_k$$

$= \varphi(x) =$ nodal polynomial

Denote **weight**

$$w_k = \prod_{j \neq k} \frac{1}{x_k - x_j}$$

$$\Rightarrow \quad P_n(x) = \varphi(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k} y_k$$

Now $\quad P_n(x_k) = y_k \quad \Rightarrow$

$$1 = \varphi(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k}$$

$$\Rightarrow \quad \varphi(x) = \frac{1}{\sum_{k} \frac{w_k}{x - x_k}}$$

$$P_n(x) = \frac{\sum_{k=0}^{n} \frac{w_k}{x - x_k} y_k}{\sum_{k=0}^{n} \frac{w_k}{x - x_k}}$$

$$w_k = \prod_{j \neq k} \frac{1}{x_k - x_j}$$ Barycentric formula

Computing the weights w_k still takes $O(n^2)$ operations but once we fix the nodes they don't change so they can be pre-computed.

Once we have w_k then we can evaluate $p_n(x)$ in $O(n)$ FLOPS, which is great.

Furthermore, it turns out the barycentric formula does not suffer from numerical roundoff error, so it is the one to use on a computer. (Not what polyfit does)

Convergence of $p(x) \to f(x)$

How good of an approximation is $p(x)$ to $f(x)$?

We need some way to measure a error, i.e., to compute a norm of $f(x) - p(x)$.

This is a nontrivial issue in functional analysis but we will start with the L_∞ norm:

$$\text{error} = \| f(x) - p(x) \|_1 = \max_{a \le x \le b} | f(x) - p(x) |$$

The usual approach to analyze error is via Taylor series, which would work if $p(x)$ were the Taylor series.

If it were, then we would have

$$f(x) - p_{\text{Taylor}}(x) = \frac{f^{n+1}(\xi)}{(n+1)!}(x - \underset{\substack{\uparrow \\ \text{Taylor} \\ \text{series origin}}}{x_0})^{n+1}$$

$$\xi \in [a,b]$$

For the interpolating polynomial, a similar estimate holds, as derived in the textbooks. For us, the formula will be sufficient:

$$f(x) - p(x) = \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}}_{\text{How smooth the function is}} \underbrace{\prod_{k=0}^{n}(x - x_k)}_{\text{Nodal polynomial}}$$

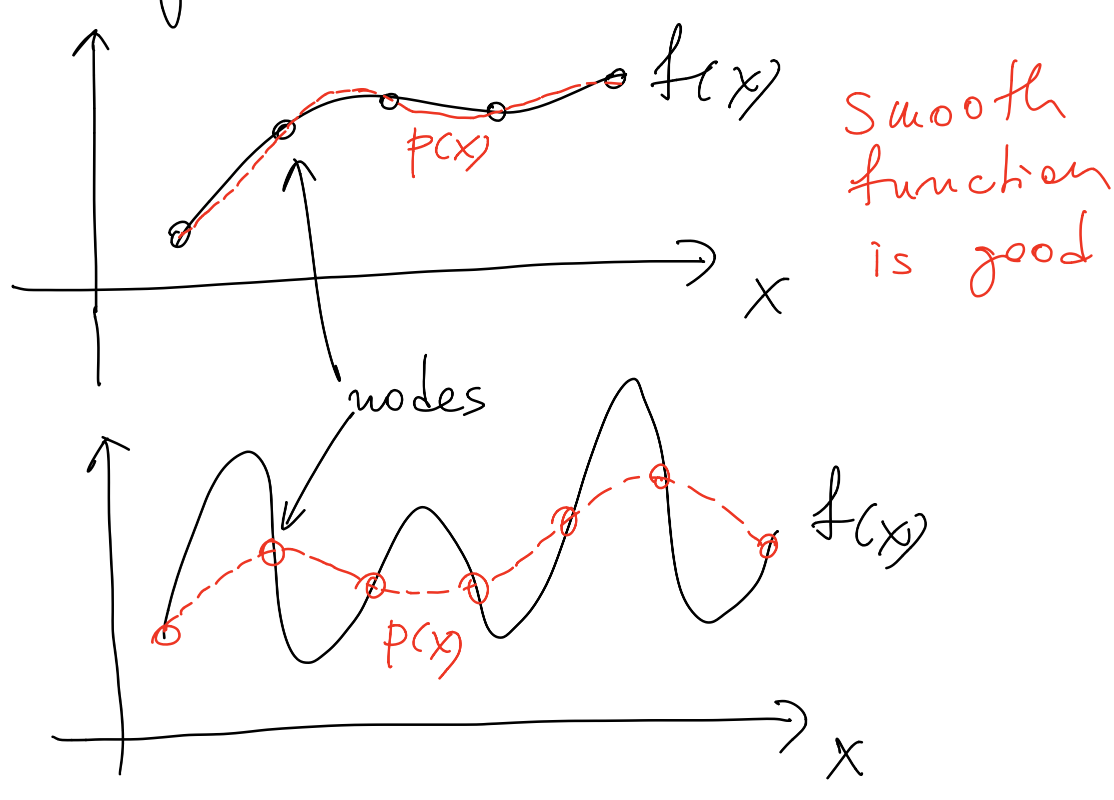
$$\xi(x) \in [a,b]$$

This formula tells us that two things matter together:

1) The magnitude of the high-order derivatives

$$|f^{(n+1)}(\xi)|, \quad \xi \in [a, b]$$

E.g.



2) The choice of the nodes dictates the nodal polynomial

$$q(x) = \prod_{k=0}^{n} (x - x_k) \quad \left(\begin{array}{c}\text{poly of} \\ \text{degree} \\ n+1\end{array}\right)$$

What matters is the magnitude of $q(x)$ (so-called Lebesgue constant (see Wiki))

$|q(x)|$ versus x

$\boxed{\text{Matlab demo}}$ NodePoly.m & RungeDemo.m

This demo illustrates a few important points (see also Worksheet)

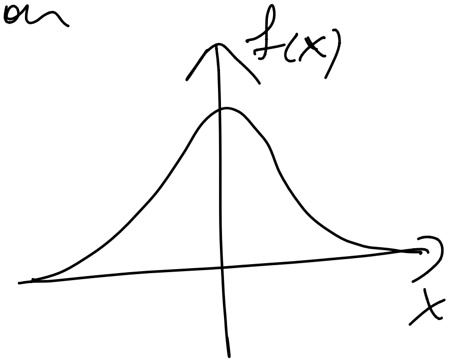
(1) For very "nice" functions like $f(x) = e^x$, the term

$$\left| \frac{f^{(n+1)}(\xi)}{(n+1)!} \right|$$

is sufficiently small and we get error that is small and converges to zero as $n \to \infty$

(2) For some seemingly smooth but not-so-nice functions like the Runge function

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



we get very large errors when we use equi-spaced nodes, and so we conclude:

Polynomial interpolants do $\underline{\underline{\text{not}}}$ converge to $f(x)$ as $n \Rightarrow \infty$ for equi-spaced nodes for generic smooth functions

This means polynomial interpolation failed our original goal of accurate function approximation if we use equi-spaced nodes.

But, the demo also showed that

If polynomial interpolation uses specially-chosen nodes that cluster near the endpoints, $p(x) \to f(x)$ as n increases.

There are many known choices of nodes that make

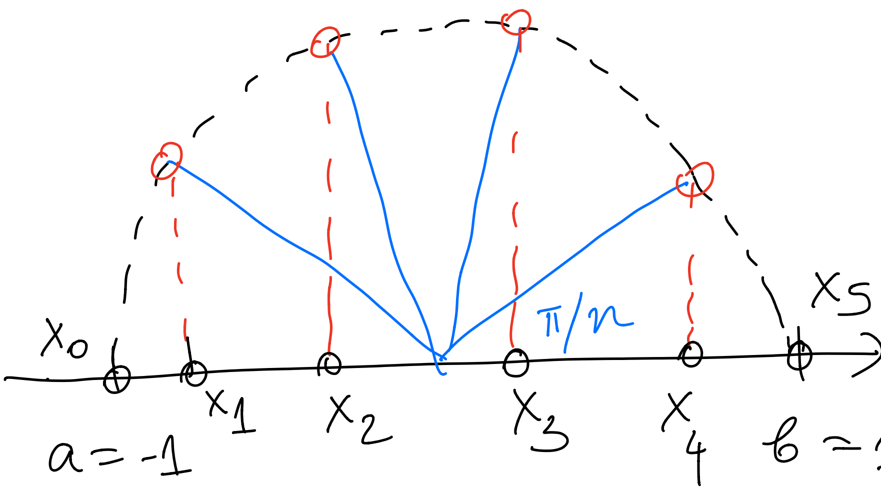
$|q(x)|$ be "small" over <u>all</u> of $[a,b]$

nodal polynomial

instead of blowing up at the endpoints (exponentially fast in n).

We will learn later of ways to come up with them but here is one choice:

Chebyshev nodes lead to accurate & robust polynomial interpolation of (sufficiently) smooth functions



$$x_k = \cos\left(\frac{\pi k}{n}\right), \quad k=0,\dots,n$$

on $[-1, 1]$ ⇐ standard interval

$$x_k = \frac{1}{2}(a+b) + \left(\frac{b-a}{2}\right) \cdot \cos\left(\frac{\pi k}{n}\right)$$

for a different interval $[a,b]$