

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.

(1)

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 $P(x)$? ②

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

$$\forall \epsilon > 0, \exists p(x) \text{ s.t.}$$

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

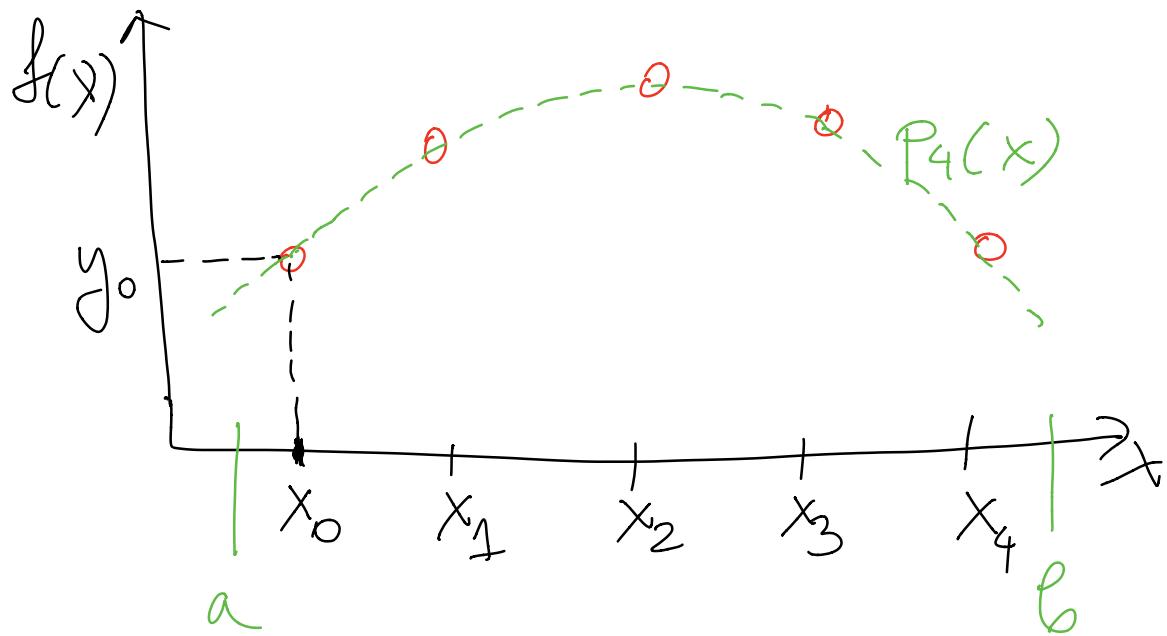
But we don't know what degree $p(x)$ is, so this doesn't help us numerically.

We want to restrict the degree of the polynomial

(3)

Interpolation

One way to go about constructing an actual polynomial $P_n(x) \approx f(x)$ 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.

(4)

$(n+1)$ 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

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

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

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

⑤

In matrix form

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

Vandermonde matrix ✓

$$\begin{array}{c} \leftrightarrow \\ V \end{array} \begin{array}{c} \rightarrow \\ a = y \end{array}$$

is a linear system we can solve using LU factorization.

Theorem: V is invertible if nodes are distinct, because

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

⑥

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 fitting a polynomial through ~~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.

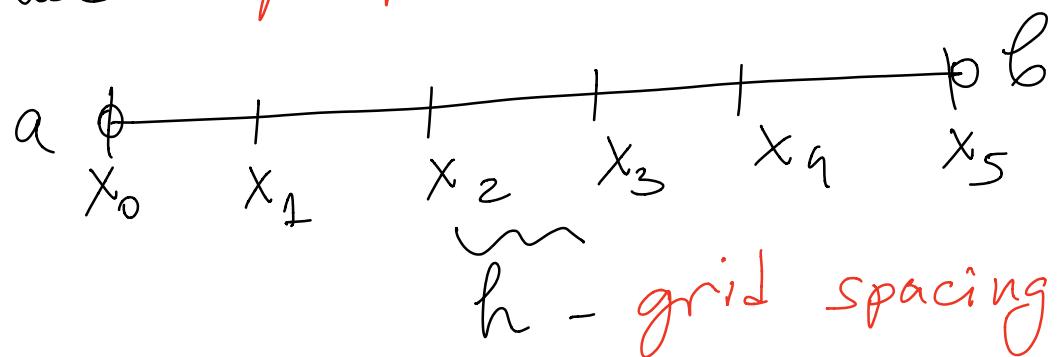
Observation: The Vandermonde matrix is generally very ill-conditioned for large n

(7)

(unless nodes are chosen carefully)

We should expect some problems !

Most obvious choice is to use **equispaced nodes**



$$\left\{ \begin{array}{l} x_i = i \cdot h + a \\ h = \frac{b-a}{n} \end{array} \right.$$

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

(8)

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 many ways to
do this, so let's discuss
a few.

The right way to think
about this is via
abstract linear algebra

⑨

{ Namely, polynomials of
 degree n form a linear
 Space P_n of dimension
 $(n+1)$.

The standard basis for this
 space are the monomials x^k

But is there a better
 basis for polynomial interpolation.

The best basis would be
 one for which

$\overset{\leftarrow}{V} \rightarrow$ identity matrix

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

Can we find this basis?

(Drop subscript n for brevity)

(10)

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

where $\{L_k\}$ is the new basis composed of polynomials of degree n . $\vec{P} = \vec{V} \vec{a}$

$$\begin{aligned} P(x_i) &= \sum a_k L_k(x_i) \\ &= V_{ik} a_k \end{aligned}$$

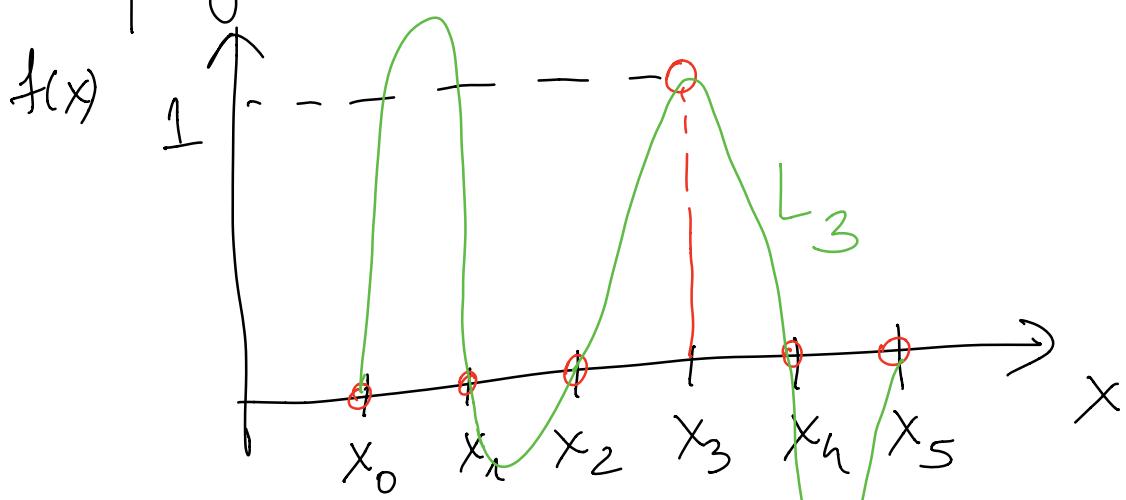
$$\Rightarrow V_{ik} = L_k(x_i) = \delta_{ik}$$

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

\uparrow
Kronecker symbol

(11)

So L_k is the interpolating polynomial of :



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

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

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

(12)

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

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

$$P_n(x) = \sum y_k L_k(x)$$

≈ f(x)

Lagrange formula for
interpolating polynomial.

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

(13)

It doesn't mean we are done, however. As good numerical analysts 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 most important: How good of an approximation of $f(x)$ is $P(x)$ for n nodes?

(14)

Question #1 : Efficiency

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

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$$
$$= a_0 + x \underbrace{(a_1 + a_2 x + \dots + a_n x^{n-1})}_{p_{n-1}(x)}$$

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

Giving :

$$\begin{cases} b_{n-1} = a_{n-1} + a_n x \\ b_{n-2} = a_{n-2} + b_{n-1} x \\ \vdots \\ b_0 = a_0 + b_1 x = p(x) \end{cases} \quad (15)$$

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 $O(n^3)$ 16

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:

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

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

Denote weight

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

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

Now $P_n(x_k) = y_k \Rightarrow$

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

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

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

(18)

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

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) ⑯

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

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

We need some way to measure 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:

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

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

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

$\xi \in [a, b]$

↑
Taylor series origin

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

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^n (x-x_k)$$

$\xi \in [a, b]$

How smooth the function is Nodal polynomial

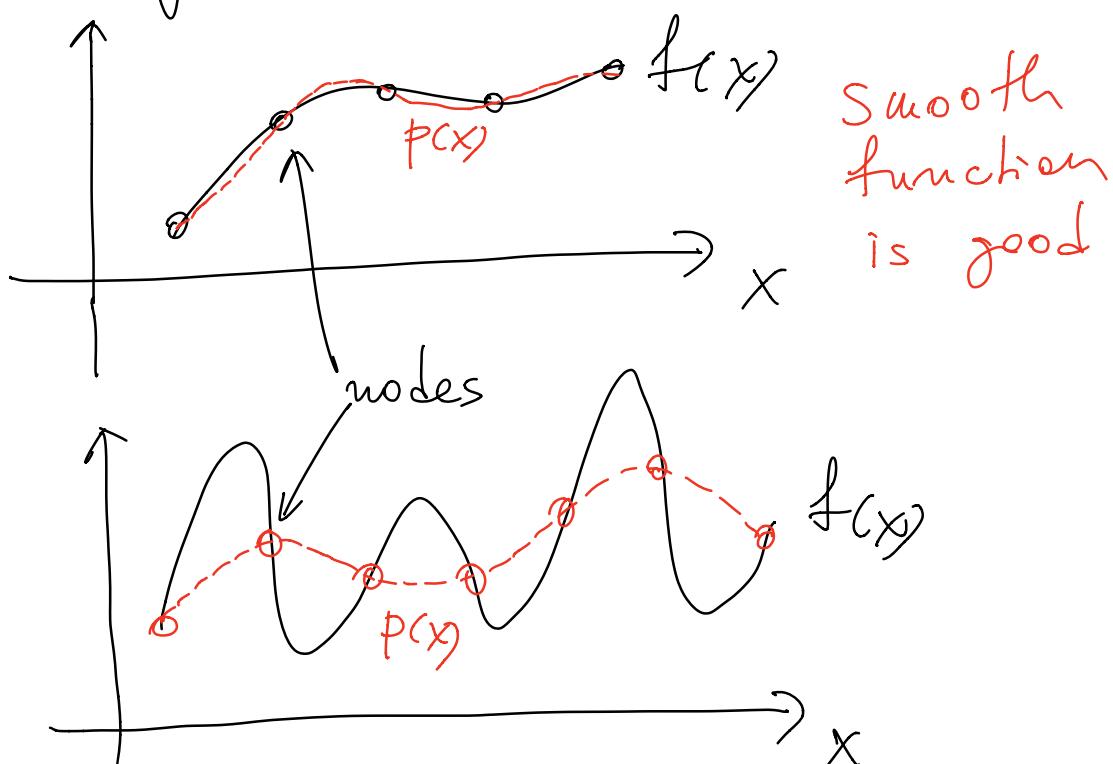
(21)

This formula tells us that two things matter together:

- 1) The magnitude of the higher order derivatives

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

E.g.



2) The choice of the nodes
dictates the nodal polynomial

$$q(x) = \prod_{k=0}^n (x - x_k)$$

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

$$|q(x)| \text{ versus } x$$

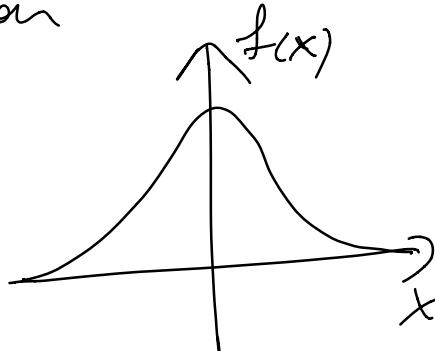
[Matlab Demos] Node Poly.m
& Runge Demos.m

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 $n \rightarrow \infty$

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

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



(24)

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

Polynomial interpolants do 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-closely nodes that cluster near the endpoints, $p(x) \rightarrow f(x)$ as n increases

There are many known choices of nodes that make

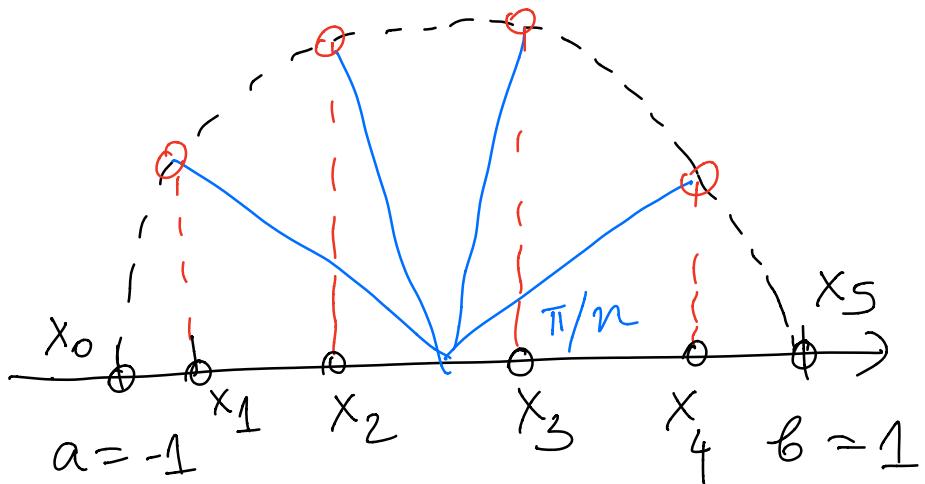
$\left| g(x) \right|$ be "small" over all of $[a, b]$
↑
nodal polynomial

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:

(26)

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



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

on $[-1, 1]$ ← standard interval

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

for a different interval $[a, b]$

(27)