

Lecture 7: Function Approximation

Zhuotong Liu

University of Edinburgh

Autumn 2025

Why Function Approximation

Consider a dynamic consumption–savings problem: an individual chooses consumption c_t and savings a_{t+1} to maximize lifetime utility (*formal solving methods coming next week*):

$$V(a_t) = \max_{c_t, a_{t+1}} \left[u(c_t) + \beta V(a_{t+1}) \right]$$

subject to the budget constraint:

$$c_t + a_{t+1} = (1 + r)a_t + y_t$$

$$c_t \geq 0, \quad a_{t+1} \geq 0$$

Interpretation:

- $V(a_t)$: value function — maximum discounted utility starting from wealth a_t
- β : discount factor, r : interest rate, y_t : income

Computational Challenge

Dynamic Programming: To solve the model by brute-force maximisation, we need $V(a, a')$ for every possible wealth and savings level a and a'

If $a \in [0, 1,000,000]$ and we use a fine grid with step size 0.1:

$$N_{a,a'} = \left(\frac{1,000,000}{0.1}\right)^2 = (10^7)^2 = 10^{14}$$

Assuming each grid point requires 8 bytes of storage and $1 \mu s$ ($10^{-6}s$) to compute, the full 2-D grid would demand 727 TB of memory and take approximately 3 years to compute

Impossible to compute or store directly!

Computational Challenge

Dynamic Programming: To solve the model by brute-force maximisation, we need $V(a, a')$ for every possible wealth and savings level a and a'

If $a \in [0, 1,000,000]$ and we use a fine grid with step size 0.1:

$$N_{a,a'} = \left(\frac{1,000,000}{0.1}\right)^2 = (10^7)^2 = 10^{14}$$

Assuming each grid point requires 8 bytes of storage and $1 \mu s$ ($10^{-6}s$) to compute, the full 2-D grid would demand 727 TB of memory and take approximately 3 years to compute

Impossible to compute or store directly!

When Function Approximation

Objective: Obtain an approximation for $f(x)$ by another $\hat{f}(x)$

Two typical scenarios:

1 Approximation

$f(x)$ is known for all x , but evaluating it is costly or difficult

Example: $f(x) = \int_0^1 e^{-t^2 x} dt$

→ expensive integral, use a polynomial $\hat{f}(x)$ instead

2 Interpolation

$f(x)$ is only known at discrete points $(x_i, f(x_i))$.

Example: we know $f(x) = \ln x$ only at sampled values, and we reconstruct the curve through them.

Key question: how can we measure the goodness of approximation; namely, how “close” $f(x)$ and $\hat{f}(x)$ are?

When Function Approximation

Objective: Obtain an approximation for $f(x)$ by another $\hat{f}(x)$

Two typical scenarios:

1 Approximation

$f(x)$ is known for all x , but evaluating it is costly or difficult

Example: $f(x) = \int_0^1 e^{-t^2 x} dt$

→ expensive integral, use a polynomial $\hat{f}(x)$ instead

2 Interpolation

$f(x)$ is only known at discrete points $(x_i, f(x_i))$.

Example: we know $f(x) = \ln x$ only at sampled values, and we reconstruct the curve through them.

Key question: how can we measure the goodness of approximation; namely, how “close” $f(x)$ and $\hat{f}(x)$ are?

When Function Approximation

Objective: Obtain an approximation for $f(x)$ by another $\hat{f}(x)$

Two typical scenarios:

1 Approximation

$f(x)$ is known for all x , but evaluating it is costly or difficult

Example: $f(x) = \int_0^1 e^{-t^2 x} dt$

→ expensive integral, use a polynomial $\hat{f}(x)$ instead

2 Interpolation

$f(x)$ is only known at discrete points $(x_i, f(x_i))$.

Example: we know $f(x) = \ln x$ only at sampled values, and we reconstruct the curve through them.

Key question: how can we measure the goodness of approximation; namely, how “close” $f(x)$ and $\hat{f}(x)$ are?

Section 1

Function Space

From Euclidean Space to Function Space

- In \mathbb{R}^n , a vector is $x = (x_1, x_2, \dots, x_n)^T$
- With addition and scalar multiplication, \mathbb{R}^n forms a **vector space**
- We can extend this idea to functions:

$$f, g : [a, b] \rightarrow \mathbb{R}$$

Define:

$$(f + g)(x) = f(x) + g(x), \quad (af)(x) = a \cdot f(x)$$

- The set of continuous functions

$$C^0([a, b], \mathbb{R})$$

is therefore a **vector space**

Intuition: Each function is like a “vector with infinitely many components” indexed by x

From Euclidean Space to Function Space

- In \mathbb{R}^n , a vector is $x = (x_1, x_2, \dots, x_n)^T$
- With addition and scalar multiplication, \mathbb{R}^n forms a **vector space**
- We can extend this idea to functions:

$$f, g : [a, b] \rightarrow \mathbb{R}$$

Define:

$$(f + g)(x) = f(x) + g(x), \quad (af)(x) = a \cdot f(x)$$

- The set of continuous functions

$$C^0([a, b], \mathbb{R})$$

is therefore a **vector space**

Intuition: Each function is like a “vector with infinitely many components” indexed by x

Norms on Function Spaces

- A **norm** measures the size (length / distance) of a function
- Common norms:

$$\|f\|_1 = \int_a^b |f(x)| dx \quad (L_1 \text{ norm})$$

$$\|f\|_2 = \left(\int_a^b |f(x)|^2 dx \right)^{1/2} \quad (L_2 \text{ norm})$$

$$\|f\|_\infty = \max_{x \in [a,b]} |f(x)| \quad (L_\infty \text{ norm})$$

- L_1, L_2, L_∞ are all normed vector spaces
- L_2 space has a geometry similar to Euclidean space

Norms on Function Spaces

- A **norm** measures the size (length / distance) of a function
- Common norms:

$$\|f\|_1 = \int_a^b |f(x)| dx \quad (L_1 \text{ norm})$$

$$\|f\|_2 = \left(\int_a^b |f(x)|^2 dx \right)^{1/2} \quad (L_2 \text{ norm})$$

$$\|f\|_\infty = \max_{x \in [a,b]} |f(x)| \quad (L_\infty \text{ norm})$$

- L_1, L_2, L_∞ are all normed vector spaces
- L_2 space has a geometry similar to Euclidean space

In Euclidean Space (\mathbb{R}^n)

- Inner product:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^n x_i y_i = \|\mathbf{x}\| \|\mathbf{y}\| \cos \theta$$

- Length (norm):

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

- Angle:

$$\cos \theta = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

- Distance:

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{\sum_i (x_i - y_i)^2}$$

In Function Space ($L_2([a, b], \mathbb{R})$)

- Inner product:

$$\langle f, g \rangle = \int_a^b f(x)g(x) dx = \|f\|_2 \|g\|_2 \cos \theta$$

- Length (norm):

$$\|f\|_2 = \sqrt{\int_a^b f(x)^2 dx}$$

- Angle:

$$\cos \theta = \frac{\langle f, g \rangle}{\|f\|_2 \|g\|_2}$$

- Distance:

$$d(f, g) = \|f - g\|_2 = \sqrt{\int_a^b (f(x) - g(x))^2 dx}$$

Why Function Spaces Matter in Approximation

- Function approximation means: find \hat{f} close to f
- But how do we measure “close”? \Rightarrow Need a norm

$$\text{Error: } \|f - \hat{f}\|_p$$

- Examples:
 - L_2 : minimize squared error \Rightarrow **Least squares approximation**
 - L_∞ : minimize maximum error \Rightarrow **Minimax approximation**
- Orthogonal polynomials (Chebyshev, Legendre) are built using the L_2 inner product

Intuition:

Function spaces provide the geometry for approximation.

They define distance, angle, and projection between functions

Why Function Spaces Matter in Approximation

- Function approximation means: find \hat{f} close to f
- But how do we measure “close”? \Rightarrow Need a norm

$$\text{Error: } \|f - \hat{f}\|_p$$

- Examples:
 - L_2 : minimize squared error \Rightarrow **Least squares approximation**
 - L_∞ : minimize maximum error \Rightarrow **Minimax approximation**
- Orthogonal polynomials (Chebyshev, Legendre) are built using the L_2 inner product

Intuition:

Function spaces provide the geometry for approximation.

They define distance, angle, and projection between functions

Why Function Spaces Matter in Approximation

- Function approximation means: find \hat{f} close to f
- But how do we measure “close”? \Rightarrow Need a norm

$$\text{Error: } \|f - \hat{f}\|_p$$

- Examples:
 - L_2 : minimize squared error \Rightarrow **Least squares approximation**
 - L_∞ : minimize maximum error \Rightarrow **Minimax approximation**
- Orthogonal polynomials (Chebyshev, Legendre) are built using the L_2 inner product

Intuition:

Function spaces provide the geometry for approximation.

They define distance, angle, and projection between functions

Section 2

Polynomial Approximation

Weierstrass Approximation Theorem

Theorem (Weierstrass, 1885)

Let f be a continuous function on $[a, b]$. Then for any $\varepsilon > 0$, there exists a polynomial $p_n(x)$ such that

$$\|f - p_n\|_\infty < \varepsilon, \quad \forall x \in [a, b].$$

- This means polynomials are universal approximators for continuous functions on a compact interval
- For sufficiently large n , $p_n(x)$ can get arbitrarily close to $f(x)$ everywhere

Weierstrass Approximation Theorem

Theorem (Weierstrass, 1885)

Let f be a continuous function on $[a, b]$. Then for any $\varepsilon > 0$, there exists a polynomial $p_n(x)$ such that

$$\|f - p_n\|_\infty < \varepsilon, \quad \forall x \in [a, b].$$

- This means polynomials are universal approximators for continuous functions on a compact interval
- For sufficiently large n , $p_n(x)$ can get arbitrarily close to $f(x)$ everywhere

Polynomial Approximation: Motivation

- We are given a continuous function $f : [a, b] \rightarrow \mathbb{R}$
- We wish to approximate it by a **simpler function**, e.g. a polynomial:

$$p_n(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n.$$

- Unlike Taylor series (local), polynomial approximation is **global** on $[a, b]$
- The set of all polynomials of degree $\leq n$ forms an $(n + 1)$ -dimensional vector space with basis $\{1, x, x^2, \dots, x^n\}$

Pros:

Polynomials are easy to evaluate, differentiate, and integrate — and can approximate any continuous function to any desired accuracy

Polynomial Approximation: Motivation

- We are given a continuous function $f : [a, b] \rightarrow \mathbb{R}$
- We wish to approximate it by a **simpler function**, e.g. a polynomial:

$$p_n(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n.$$

- Unlike Taylor series (local), polynomial approximation is **global** on $[a, b]$
- The set of all polynomials of degree $\leq n$ forms an $(n + 1)$ -dimensional vector space with basis $\{1, x, x^2, \dots, x^n\}$

Pros:

Polynomials are easy to evaluate, differentiate, and integrate — and can approximate any continuous function to any desired accuracy

Lagrange Interpolation

- Choose $(n + 1)$ distinct points (nodes): x_0, x_1, \dots, x_n .
- Require the polynomial to pass through these points:

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

- This gives $(n + 1)$ linear equations:

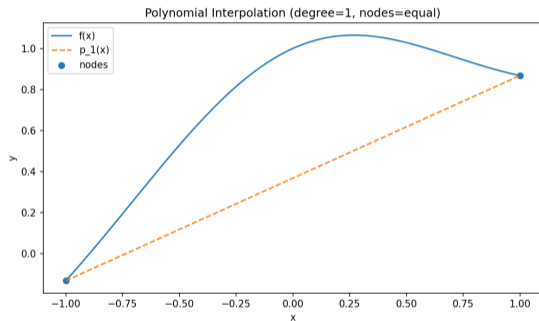
$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}}_X \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}$$

- If all x_i are distinct, X (Vandermonde matrix) is invertible:

$$\mathbf{c} = X^{-1}\mathbf{y}.$$

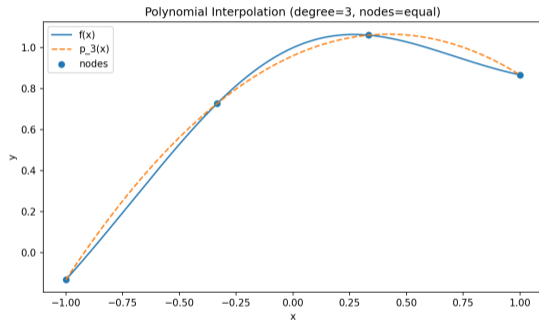
Lagrange Interpolation in Practice

```
def vandermonde(x_nodes, y_nodes):  
    n = len(x_nodes) - 1  
    X = np.vander(x_nodes, N=n+1, increasing=True)  
    c = np.linalg.solve(X, y_nodes)  
    return c  
  
def poly_eval(c, x):  
    y = np.zeros_like(x, dtype=float)  
    for i, coef in enumerate(c):  
        y += coef * x**i  
    return y
```



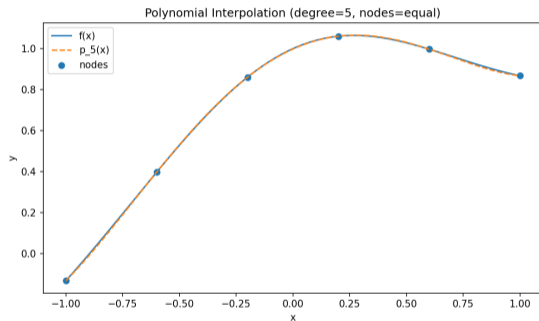
Lagrange Interpolation in Practice

```
def vandermonde(x_nodes, y_nodes):  
    n = len(x_nodes) - 1  
    X = np.vander(x_nodes, N=n+1, increasing=True)  
    c = np.linalg.solve(X, y_nodes)  
    return c  
  
def poly_eval(c, x):  
    y = np.zeros_like(x, dtype=float)  
    for i, coef in enumerate(c):  
        y += coef * x**i  
    return y
```



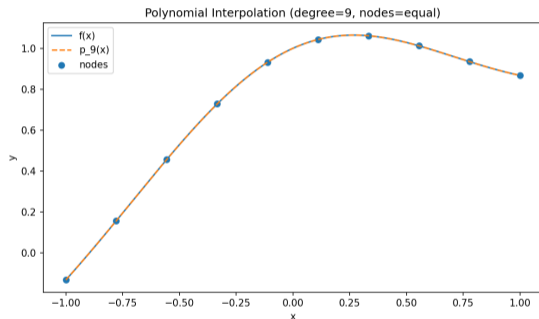
Lagrange Interpolation in Practice

```
def vandermonde(x_nodes, y_nodes):  
    n = len(x_nodes) - 1  
    X = np.vander(x_nodes, N=n+1, increasing=True)  
    c = np.linalg.solve(X, y_nodes)  
    return c  
  
def poly_eval(c, x):  
    y = np.zeros_like(x, dtype=float)  
    for i, coef in enumerate(c):  
        y += coef * x**i  
    return y
```



Lagrange Interpolation in Practice

```
def vandermonde(x_nodes, y_nodes):  
    n = len(x_nodes) - 1  
    X = np.vander(x_nodes, N=n+1, increasing=True)  
    c = np.linalg.solve(X, y_nodes)  
    return c  
  
def poly_eval(c, x):  
    y = np.zeros_like(x, dtype=float)  
    for i, coef in enumerate(c):  
        y += coef * x**i  
    return y
```



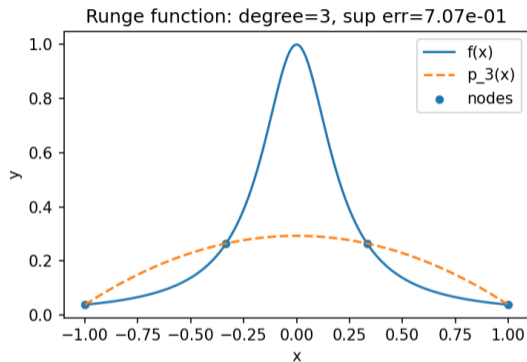
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



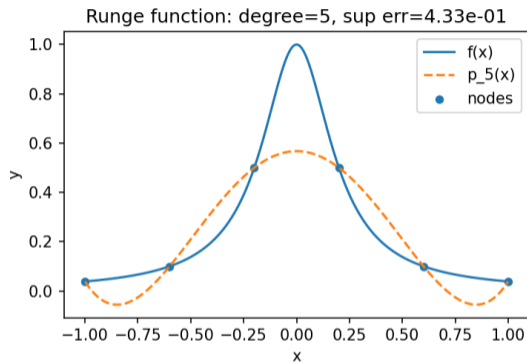
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



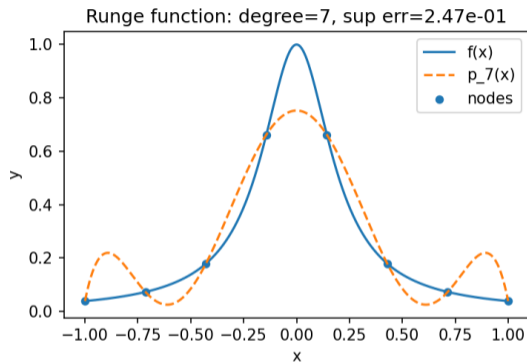
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



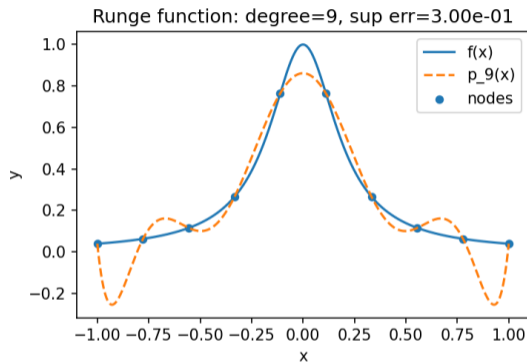
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



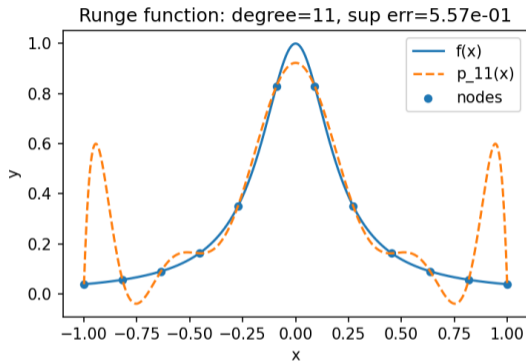
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



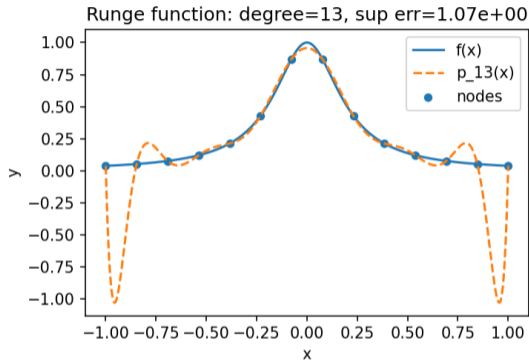
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



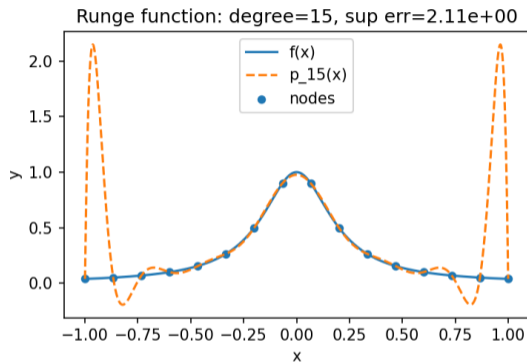
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



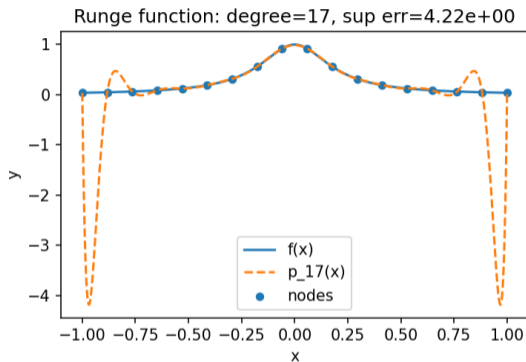
Lagrange Interpolation may fail badly

- So far, it seems Lagrange interpolation works well
- However, there are some well-known cases where it oscillates near endpoints

- Consider

$$f(x) = \frac{1}{1 + 25x^2}$$

- Lagrange interpolation will never converge in this case
- Adding more data doesn't fix this problem. This is called **Runge Phenomenon**
- It is because equally spaced nodes result in the high-degree instability



How to avoid the Runge phenomenon

- Since the Runge phenomenon tends to occur in most polynomial interpolation schemes with equally spaced nodes, to avoid it,
 - ① **Splines:** use piecewise low-degree polynomials and enforce smoothness at knots
 - ② **Chebyshev polynomials:** space more points near the interval endpoints to reduce oscillations

Section 3

Spline Interpolation

Spline Interpolation: Local Smooth Approximation

Motivation: Approximate f on small subintervals by low-order polynomials and join them smoothly

Definition: Divide $[a, b]$ into n subintervals

$$a = x_0 < x_1 < \cdots < x_n = b.$$

On each $[x_i, x_{i+1}]$, approximate f locally by a low-order polynomial

$$p_i(x) = c_{i0} + c_{i1}(x - x_i) + c_{i2}(x - x_i)^2 + \cdots + c_{im}(x - x_i)^m, \quad x \in [x_i, x_{i+1}],$$

and define the spline

$$S(x) = \begin{cases} p_0(x), & x \in [x_0, x_1], \\ p_1(x), & x \in [x_1, x_2], \\ \vdots & \\ p_{n-1}(x), & x \in [x_{n-1}, x_n]. \end{cases}$$

Continuity constraints (for cubic splines): At each interior knot x_i :

$$p_{i-1}(x_i) = p_i(x_i) \quad (\text{function continuous}),$$

$$p'_{i-1}(x_i) = p'_i(x_i) \quad (\text{first derivative continuous}),$$

$$p''_{i-1}(x_i) = p''_i(x_i) \quad (\text{second derivative continuous}).$$

Spline Interpolation: Local Smooth Approximation

Definition: Divide $[a, b]$ into n subintervals

$$a = x_0 < x_1 < \cdots < x_n = b.$$

On each $[x_i, x_{i+1}]$, approximate f locally by a low-order polynomial

$$p_i(x) = c_{i0} + c_{i1}(x - x_i) + c_{i2}(x - x_i)^2 + \cdots + c_{im}(x - x_i)^m, \quad x \in [x_i, x_{i+1}],$$

and define the spline

$$S(x) = \begin{cases} p_0(x), & x \in [x_0, x_1], \\ p_1(x), & x \in [x_1, x_2], \\ \vdots & \\ p_{n-1}(x), & x \in [x_{n-1}, x_n]. \end{cases}$$

Continuity constraints (for cubic splines): At each interior knot x_i :

$$p_{i-1}(x_i) = p_i(x_i) \quad (\text{function continuous}),$$

$$p'_{i-1}(x_i) = p'_i(x_i) \quad (\text{first derivative continuous}),$$

$$p''_{i-1}(x_i) = p''_i(x_i) \quad (\text{second derivative continuous}).$$

Linear Spline Interpolation

Definition: Approximate $f : [a, b] \rightarrow \mathbb{R}$ by piecewise linear functions on subintervals

$$a = x_0 < x_1 < \cdots < x_n = b.$$

On each $[x_i, x_{i+1}]$, define

$$p_i(x) = a_i + b_i(x - x_i), \quad x \in [x_i, x_{i+1}].$$

Conditions: We require

$$\begin{cases} p_i(x_i) = f(x_i) = y_i, & i = 0, \dots, n-1, \\ p_i(x_{i+1}) = p_{i+1}(x_{i+1}), & i = 0, \dots, n-2. \end{cases}$$

Linear Spline Interpolation

Linear System Representation:

For n subintervals:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & h_0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & h_1 & \cdots & 0 \\ \vdots & & & & \ddots & \\ 0 & \cdots & 0 & 0 & 1 & h_{n-1} \end{bmatrix}}_{2n \times 2n} \underbrace{\begin{bmatrix} a_0 \\ b_0 \\ a_1 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}}_{\mathbf{c}} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}}, \quad h_i = x_{i+1} - x_i.$$

Closed-form Solution:

$$a_i = y_i, \quad b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

Thus,

$$p_i(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i} (x - x_i).$$

Cubic Spline Interpolation

Definition: On each subinterval $[x_i, x_{i+1}]$,

$$p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad i = 0, \dots, n - 1.$$

Conditions:

- 1 Interpolation: $p_i(x_i) = y_i$
- 2 Function continuity: $p_i(x_{i+1}) = p_{i+1}(x_{i+1})$
- 3 First-derivative continuity: $p_i'(x_{i+1}) = p_{i+1}'(x_{i+1})$
- 4 Second-derivative continuity: $p_i''(x_{i+1}) = p_{i+1}''(x_{i+1})$

Cubic Spline Interpolation

Tri-diagonal system (natural case):

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ h_0 & 2(h_0 + h_1) & h_1 & \cdots & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & \cdots & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_n \end{bmatrix} = 6 \begin{bmatrix} 0 \\ \frac{y_2 - y_1}{h_1} - \frac{y_1 - y_0}{h_0} \\ \vdots \\ 0 \end{bmatrix}, \quad h_i = x_{i+1} - x_i.$$

Closed-form Solution:

$$p_i(x) = \frac{M_i(x_{i+1} - x)^3}{6h_i} + \frac{M_{i+1}(x - x_i)^3}{6h_i} + \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6} \right) (x_{i+1} - x) + \left(\frac{y_{i+1}}{h_i} - \frac{M_{i+1} h_i}{6} \right) (x - x_i).$$

Cubic Spline Interpolation and Boundary Conditions

Interior continuity conditions (for all $i = 0, \dots, n - 2$):

$$p_i(x_{i+1}) = p_{i+1}(x_{i+1}) \quad (\text{function})$$

$$p'_i(x_{i+1}) = p'_{i+1}(x_{i+1}) \quad (\text{first derivative})$$

$$p''_i(x_{i+1}) = p''_{i+1}(x_{i+1}) \quad (\text{second derivative})$$

These provide $4n - 2$ equations for $4n$ unknowns \Rightarrow two boundary conditions needed

Cubic Spline Interpolation and Boundary Conditions

Common boundary conditions:

1 Natural spline:

$$p_0''(x_0) = 0, \quad p_{n-1}''(x_n) = 0.$$

Minimizes bending energy; smooth, “free-en” condition

2 Hermite (Clamped) spline:

$$p_0'(x_0) = f'(x_0), \quad p_{n-1}'(x_n) = f'(x_n).$$

Forces given endpoint slopes

3 Secant spline:

$$p_0'(x_0) = \frac{y_1 - y_0}{x_1 - x_0}, \quad p_{n-1}'(x_n) = \frac{y_n - y_{n-1}}{x_n - x_{n-1}}.$$

Uses endpoint slopes estimated by first and last secants

4 Not-a-knot spline:

$$p_0'''(x_1) = p_1'''(x_1), \quad p_{n-2}'''(x_{n-1}) = p_{n-1}'''(x_{n-1}).$$

Treats x_1 and x_{n-1} as “non-knots”, giving highest global smoothness

Section 4

Chebyshev Orthogonal Interpolation

Least-Squares Approximation and the Gram Matrix

Setup: Approximate f on $[-1, 1]$ by basis functions ϕ_0, \dots, ϕ_n :

$$f(x) \approx \sum_{k=0}^n a_k \phi_k(x) \iff f \approx \mathbf{\Phi a}, \quad \mathbf{\Phi}(x) = [\phi_0(x), \dots, \phi_n(x)].$$

Least-squares problem:

$$\min_{\mathbf{a}} \|f - \mathbf{\Phi a}\|_2^2 = \int_{-1}^1 (f(x) - \sum a_k \phi_k(x))^2 dx.$$

Normal equations:

$$\mathbf{G a} = \mathbf{b}, \quad G_{jk} = \int_{-1}^1 \phi_j(x) \phi_k(x) dx, \quad b_j = \int_{-1}^1 f(x) \phi_j(x) dx.$$

Non-orthogonal basis (e.g. monomials $1, x, x^2, \dots$):

$$\mathbf{G} = \begin{bmatrix} \langle \phi_0, \phi_0 \rangle & \langle \phi_0, \phi_1 \rangle & \cdots \\ \langle \phi_1, \phi_0 \rangle & \langle \phi_1, \phi_1 \rangle & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

Dense, ill-conditioned \Rightarrow must solve all a_k jointly by inversion

Orthogonal Basis: Why It Simplifies the Computation

Orthogonality condition:

$$\langle \phi_j, \phi_k \rangle = \int_{-1}^1 \phi_j(x) \phi_k(x) dx = \begin{cases} 0, & j \neq k, \\ c_j > 0, & j = k. \end{cases}$$

Then the Gram matrix becomes diagonal:

$$\mathbf{G} = \begin{bmatrix} c_0 & 0 & 0 & \cdots \\ 0 & c_1 & 0 & \cdots \\ 0 & 0 & c_2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}, \quad \mathbf{G}\mathbf{a} = \mathbf{b} \Rightarrow a_j = \frac{b_j}{c_j}.$$

Coefficient formula (scalar form):

$$a_j = \frac{\int_{-1}^1 f(x) \phi_j(x) dx}{\int_{-1}^1 \phi_j^2(x) dx}.$$

Interpretation:

- Each coefficient a_j depends only on one inner product

Chebyshev Orthogonal Polynomials

Chebyshev polynomials $T_n(x)$ on $[-1, 1]$:

$$T_n(x) = \cos(n \arccos x), \quad T_0 = 1, \quad T_1 = x, \quad T_{n+1} = 2x T_n - T_{n-1}.$$

Weighted orthogonality (weight $w(x) = 1/\sqrt{1-x^2}$):

$$\int_{-1}^1 \frac{T_m(x) T_n(x)}{\sqrt{1-x^2}} dx = \begin{cases} 0, & m \neq n, \\ \pi, & n = m = 0, \\ \pi/2, & n = m > 0. \end{cases}$$

Chebyshev expansion (continuous coefficients): For $f \in L_w^2[-1, 1]$,

$$f(x) \approx \sum_{k=0}^n a_k T_k(x), \quad a_k = \frac{2}{\pi c_k} \int_{-1}^1 \frac{f(x) T_k(x)}{\sqrt{1-x^2}} dx,$$

where $c_0 = 2$ and $c_k = 1$ for $k \geq 1$.

Why this is simple. Orthogonality gives a_k in closed form—each a_k uses a single weighted inner product; no coupled system, no ill-conditioned Vandermonde

Chebyshev Interpolation

- Chebyshev interpolation works very well if the function is defined everywhere and smooth
- The smoother the function, the faster the Chebyshev approximation converges
- Sometimes it struggles at the boundary
 - Can be fixed by using a different set of points x_i that include the boundary node
 - This is called the **expanded Chebyshev array**
- Problems occur when the function is not bounded
 - For example, a utility function that goes to $-\infty$ as $c \rightarrow 0$ can cause numerical issues
- Functions with kinks or discontinuous derivatives lose all convergence guarantees