March 25, 2026

Contents

List of Algorithms

Preface

Numerical analysis is the art of solving mathematical problems *approximately but with controlled error*. Unlike theoretical analysis, which establishes existence and uniqueness of solutions, numerical analysis is concerned with *actually computing* them and *quantifying the error incurred*.

This course covers the fundamentals: floating-point arithmetic, solving linear systems, interpolation, numerical integration, ordinary differential equations, and eigenvalue problems. Each method is accompanied by its error analysis, pseudocode, and implementations in Python and Julia.

Prerequisites. Real Analysis I & II, linear algebra, basic programming.

References.

- Quarteroni, Sacco, Saleri — *Numerical Mathematics*, Springer.
- Süli, Mayers — *An Introduction to Numerical Analysis*, Cambridge.
- Trefethen, Bau — *Numerical Linear Algebra*, SIAM.
- Higham — *Accuracy and Stability of Numerical Algorithms*, SIAM.
- Demailly — *Analyse Numérique et Équations Différentielles*, EDP Sciences.

Chapter 1

Floating-Point Arithmetic, Errors and Stability

"Numerical computation is the art of giving the right answer to a slightly different problem."

1.1 Motivating example

Let us compute $f(x) = \frac{1-\cos x}{x^2}$ for $x = 10^{-8}$ in double precision. The theoretical result is ≈ 0.5, but Python returns 0.0! The **catastrophic cancellation** between 1 and $\cos x$ eliminates all significant digits. Understanding floating-point arithmetic is essential.

1.2 Representation of numbers in a computer

Definition 1.1 (Floating-point number)**.** In base β with t significant digits, a floating-point number is written:

$$x = \pm m \times \beta^e, \quad m = d_0.d_1 d_2 \cdots d_{t-1}, \quad d_0 \neq 0,$$

where $e_{\min} \leq e \leq e_{\max}$ and $0 \leq d_i < \beta$.

Definition 1.2 (IEEE 754 Standard)**.**

Format	**Bits**	**Mantissa**	$\varepsilon_{\text{mach}}$
Single precision	32	23+1 bits	$\approx 1.19 \times 10^{-7}$
Double precision	64	52+1 bits	$\approx 2.22 \times 10^{-16}$

Definition 1.3 (Machine epsilon)**.** The **machine epsilon** $\varepsilon_{\text{mach}}$ is the smallest $\varepsilon > 0$ such that $\text{fl}(1+\varepsilon) > 1$:

$$\varepsilon_{\text{mach}} = \frac{1}{2}\beta^{1-t}.$$

For every representable $x \in \mathbb{R}$, $\text{fl}(x) = x(1+\delta)$ with $|\delta| \leq \varepsilon_{\text{mach}}$.

1.3 Rounding errors and propagation

Definition 1.4 (Absolute and relative error)**.**

$$\text{Absolute error} : E_a = |x - \tilde{x}|, \tag{1.1}$$

$$\text{Relative error} : E_r = \frac{|x - \tilde{x}|}{|x|} \quad (x \neq 0). \tag{1.2}$$

Theorem 1.5 (Error propagation)**.** *If $\tilde{x} = x(1 + \varepsilon_x)$ and $\tilde{y} = y(1 + \varepsilon_y)$:*

- $\tilde{x} \cdot \tilde{y} \approx xy(1 + \varepsilon_x + \varepsilon_y)$*: relative errors add up.*
- $\tilde{x} - \tilde{y} \approx (x - y) + x\varepsilon_x - y\varepsilon_y$*: if $x \approx y$, the relative error blows up.*

Warning

Catastrophic cancellation occurs when subtracting nearly equal numbers. Example: $1 - \cos x$ for small x. Remedy: use $2\sin^2(x/2)$.

1.4 Conditioning of a problem

Definition 1.6 (Condition number)**.** The **condition number** of a problem f at the point x is:

$$\kappa(f, x) = \left| \frac{x f'(x)}{f(x)} \right|.$$

A problem is **well-conditioned** if $\kappa \sim 1$, **ill-conditioned** if $\kappa \gg 1$.

Definition 1.7 (Matrix condition number)**.** For $Ax = b$:

$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\|.$$

The relative error on x is amplified by $\text{cond}(A)$:

$$\frac{\|\delta x\|}{\|x\|} \leq \text{cond}(A) \frac{\|\delta b\|}{\|b\|}.$$

1.5 Numerical stability

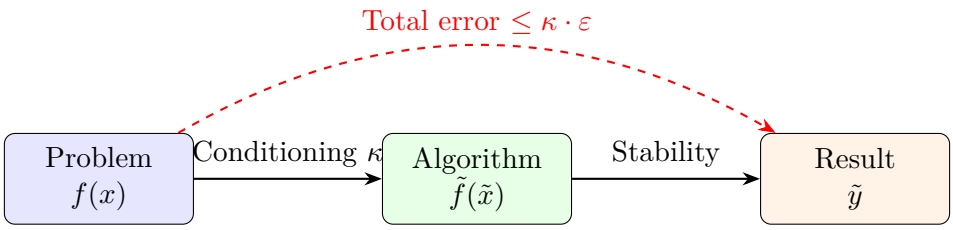
Definition 1.8 (Stable algorithm)**.** An algorithm is **stable** if it gives the exact solution of a **slightly perturbed** problem. It is **backward stable** if the perturbation is of order $\varepsilon_{\text{mach}}$.

Method

The accuracy of the result depends on two factors:

1. The **conditioning** of the problem (inherent, cannot be changed).
2. The **stability** of the algorithm (the programmer's choice).

Rule: error $\lesssim$ cond $\times\, \varepsilon_{\text{mach}}$.



1.6 Implementation

Python

```python
import numpy as np

# Epsilon machine
eps = np.finfo(float).eps
print(f"eps_mach = {eps}")  # 2.220446049250313e-16

# Cancellation catastrophique
x = 1e-8
f_bad = (1 - np.cos(x)) / x**2
f_good = 2 * np.sin(x/2)**2 / x**2
print(f"Mauvais : {f_bad}")   # 0.0
print(f"Bon     : {f_good}")  # 0.49999999...

# Conditionnement d'une matrice
A = np.array([[1, 1], [1, 1.0001]])
print(f"cond(A) = {np.linalg.cond(A):.1f}")  # ~40000
```

Julia

```julia
# Epsilon machine
println("eps_mach = ", eps(Float64))

# Cancellation
x = 1e-8
f_bad = (1 - cos(x)) / x^2
f_good = 2sin(x/2)^2 / x^2
println("Mauvais : $f_bad")
println("Bon     : $f_good")

# Conditionnement
using LinearAlgebra
A = [1.0 1.0; 1.0 1.0001]
println("cond(A) = ", cond(A))
```

1.7 Exercises

Exercise 1.1 ($\star$ – Machine epsilon)**.** Write a program that computes $\varepsilon_{\text{mach}}$ by bisection. Compare with `np.finfo(float).eps`.

Exercise 1.2 ($\star$ – Cancellation)**.** Propose a stable computation of $\sqrt{x+1} - \sqrt{x}$ for large x. Verify numerically.

Exercise 1.3 ($\star\star$ – Condition number)**.** Compute the condition number of the problem $f(x) = \ln x$ at the point $x = 1$. Interpret.

Exercise 1.4 ($\star\star$ – Hilbert matrix)**.** Let H_n be the Hilbert matrix ($h_{ij} = 1/(i+j-1)$). Compute $\text{cond}(H_n)$ for $n = 2, \ldots, 15$. What do you observe?

Exercise 1.5 ($\star\star\star$ – Project)**.** Study the numerical stability of evaluating the polynomial $p(x) = \sum a_i x^i$ using Horner's algorithm vs. naive evaluation. Compare relative errors for high-degree polynomials.

Key Formulas

$$\text{fl}(x) = x(1+\delta),\ |\delta| \le \varepsilon_{\text{mach}} \qquad \varepsilon_{\text{mach}} = \tfrac{1}{2}\beta^{1-t}$$
$$\kappa(f,x) = |xf'(x)/f(x)| \qquad \text{cond}(A) = \|A\| \cdot \|A^{-1}\|$$

Chapter 2

Root Finding

2.1 Motivating example

Finding x such that $e^x = 3x$ amounts to finding a root of $f(x) = e^x - 3x$. No closed-form solution exists: a numerical method is needed.

2.2 Bisection method

Theorem 2.1 (Intermediate Value Theorem)**.** *If $f \in C([a,b])$ and $f(a)f(b) < 0$, then $\exists\, c \in (a,b)$ such that $f(c) = 0$.*

Algorithm 1: Bisection

Input: f, a, b, tolerance ε
Output: Approximation of the root c
while $b - a > \varepsilon$ **do**
  $c \leftarrow (a+b)/2$;
  if $f(a) \cdot f(c) < 0$ **then**
    $b \leftarrow c$
  else
    $a \leftarrow c$
return c

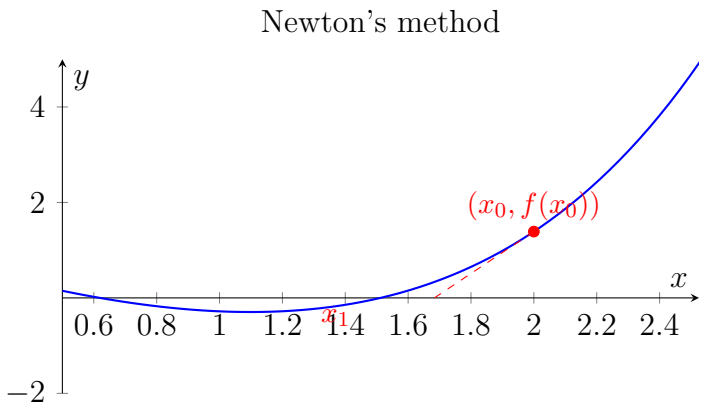
Theorem 2.2 (Convergence of bisection)**.** *After n iterations: $|c_n - x^*| \leq \frac{b-a}{2^{n+1}}$. The convergence is **linear** with rate $1/2$.*

Proof. At each iteration, the interval is halved. After n iterations, $|I_n| = (b-a)/2^n$ and $|c_n - x^*| \leq |I_n|/2$. □

2.3 Newton's method

Definition 2.3. Newton's iteration for $f(x) = 0$ is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$



Algorithm 2: Newton's method

Input: f, f', x_0, tolerance ε, $N_{\max}$
Output: Approximation x_n
for $n = 0, 1, \ldots, N_{\max}$ **do**
  $x_{n+1} \leftarrow x_n - f(x_n)/f'(x_n)$;
  if $|x_{n+1} - x_n| < \varepsilon$ **then**
    return x_{n+1}

Theorem 2.4 (Quadratic convergence of Newton's method)**.** *If $f \in C^2$, $f(x^*) = 0$, $f'(x^*) \neq 0$ and x_0 is sufficiently close to x^*, then:*

$$|e_{n+1}| \leq \frac{M}{2m}|e_n|^2,$$

where $e_n = x_n - x^*$, $M = \sup|f''|$, $m = \inf|f'|$ in a neighbourhood of x^*.

Proof. By Taylor's theorem: $0 = f(x^*) = f(x_n) + f'(x_n)(x^* - x_n) + \frac{f''(\xi_n)}{2}(x^* - x_n)^2$. Dividing by $f'(x_n)$:

$$0 = \frac{f(x_n)}{f'(x_n)} + (x^* - x_n) + \frac{f''(\xi_n)}{2f'(x_n)}e_n^2.$$

Since $x_{n+1} = x_n - f(x_n)/f'(x_n)$, we get $e_{n+1} = x_{n+1} - x^* = -\frac{f''(\xi_n)}{2f'(x_n)}e_n^2$. □

2.4 Secant method

Definition 2.5. We replace $f'(x_n)$ by a finite difference:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

Theorem 2.6. *The order of convergence of the secant method is the* ***golden ratio*** $\varphi = (1 + \sqrt{5})/2 \approx 1.618$.

2.5 Fixed-point method

Definition 2.7. Find $x = g(x)$. Iteration: $x_{n+1} = g(x_n)$.

Theorem 2.8 (Banach Fixed-Point Theorem)**.** *If $g : [a, b] \to [a, b]$ is a contraction ($|g'(x)| \leq L < 1$ on $[a, b]$), then g has a unique fixed point x^* and $x_n \to x^*$ with:*

$$|x_n - x^*| \leq \frac{L^n}{1 - L}|x_1 - x_0|.$$

2.6 Comparison of methods

Method	Order	Evaluations of f	Robustness
Bisection	1 (linear)	1/iter.	Very robust
Newton	2 (quadratic)	$f + f'$/iter.	Sensitive to x_0
Secant	$\varphi \approx 1.618$	1/iter.	Moderate

2.7 Numerical example

Root of $f(x) = e^x - 3x$ with $x_0 = 2$ (Newton):

n	x_n	$f(x_n)$	$\lvert x_n - x_{n-1}\rvert$
0	2.000000	1.389056	—
1	1.373418	0.225698	0.627
2	1.524247	-0.046	0.151
3	1.512127	-0.000374	0.012
4	1.512135	$< 10^{-9}$	8×10^{-6}

2.8 Implementation

Python

```python
import numpy as np

def bisection(f, a, b, tol=1e-12):
    while b - a > tol:
        c = (a + b) / 2
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return (a + b) / 2

def newton(f, df, x0, tol=1e-12, maxiter=100):
    x = x0
    for i in range(maxiter):
        dx = f(x) / df(x)
        x -= dx
        if abs(dx) < tol:
            return x, i+1
    return x, maxiter

def secant(f, x0, x1, tol=1e-12, maxiter=100):
    for i in range(maxiter):
        fx0, fx1 = f(x0), f(x1)
        x2 = x1 - fx1 * (x1 - x0) / (fx1 - fx0)
        if abs(x2 - x1) < tol:
            return x2, i+1
```

```
        x0, x1 = x1, x2
    return x1, maxiter

# Test
f = lambda x: np.exp(x) - 3*x
df = lambda x: np.exp(x) - 3

print(f"Bissection : {bisection(f, 0, 1):.12f}")
x_newton, n = newton(f, df, 2.0)
print(f"Newton     : {x_newton:.12f} ({n} iterations)")
x_sec, n = secant(f, 0.0, 1.0)
print(f"Secante    : {x_sec:.12f} ({n} iterations)")
```

Julia

```
function newton(f, df, x0; tol=1e-12, maxiter=100)
    x = x0
    for i in 1:maxiter
        dx = f(x) / df(x)
        x -= dx
        abs(dx) < tol && return x, i
    end
    return x, maxiter
end

f(x) = exp(x) - 3x
df(x) = exp(x) - 3
x, n = newton(f, df, 2.0)
println("Newton: x = $x ($n iterations)")
```

2.9 Exercises

Exercise 2.1 ($\star$)**.** Find $\sqrt{2}$ by applying Newton's method to $f(x) = x^2 - 2$. How many iterations are needed for 15 decimal digits?

Exercise 2.2 ($\star\star$)**.** Show that Newton's method applied to $f(x) = x^2$ converges linearly (double root). Generalise: what happens when $f'(x^*) = 0$?

Exercise 2.3 ($\star\star$)**.** Prove that the order of the secant method is φ by setting $e_{n+1} \approx C\, e_n^p$ and solving $p^2 - p - 1 = 0$.

Exercise 2.4 ($\star\star\star$ – Project)**.** Implement Brent's method (combination of bisection + secant). Compare with the simpler methods on 10 test functions.

Key Formulas

$$\text{Bisection} : |e_n| \leq \frac{b-a}{2^{n+1}}$$

$$\text{Newton} : x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad |e_{n+1}| \leq C|e_n|^2$$

$$\text{Secant} : x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \quad \text{order } \varphi$$

Chapter 3

Direct Methods for Linear Systems

3.1 Motivating example

An electrical circuit with n nodes produces a system $Ax = b$ of n equations. For $n = 1000$, a systematic and efficient method is needed.

3.2 Gaussian elimination and LU factorisation

Definition 3.1 (LU factorisation)**.** Find L lower triangular (with $\ell_{ii} = 1$) and U upper triangular such that $A = LU$. Then $Ax = b$ is solved by $Ly = b$ (forward substitution) followed by $Ux = y$ (back substitution).

Algorithm 3: LU factorisation (without pivoting)

Input: Matrix $A \in \mathbb{R}^{n \times n}$
Output: L, U such that $A = LU$
for $k = 1, \ldots, n-1$ **do**
  for $i = k+1, \ldots, n$ **do**
    $\ell_{ik} \leftarrow a_{ik}/a_{kk}$;
    for $j = k+1, \ldots, n$ **do**
      $a_{ij} \leftarrow a_{ij} - \ell_{ik} \cdot a_{kj}$;
$U \leftarrow$ upper triangular part of A;

Theorem 3.2 (Existence of LU)**.** *If all leading principal submatrices of A are invertible, then the LU factorisation exists and is unique.*

Theorem 3.3 (Cost)**.** *The LU factorisation costs $\frac{2n^3}{3} + \mathcal{O}(n^2)$ floating-point operations.*

Proof. Step k performs $(n-k)^2$ multiplications and subtractions. Total: $\sum_{k=1}^{n-1} 2(n-k)^2 = 2\sum_{j=1}^{n-1} j^2 = \frac{2n^3}{3} + \mathcal{O}(n^2)$. □

3.3 Partial pivoting

Warning

Without pivoting, the algorithm can be unstable (division by a small pivot). **Partial pivoting** permutes rows to maximise $|a_{kk}|$.

One obtains $PA = LU$ where P is a permutation matrix.

Theorem 3.4 (Stability)**.** *With partial pivoting, Gaussian elimination is* ***backward stable****:*

$$(A + \Delta A)\hat{x} = b, \quad \|\Delta A\| \leq g(n)\, \varepsilon_{\text{mach}}\, \|A\|,$$

where $g(n)$ is the growth factor (bounded by 2^{n-1} in theory, rarely exceeded in practice).

3.4 Cholesky factorisation

Theorem 3.5 (Cholesky factorisation)**.** *If A is* ***symmetric positive definite*** *(SPD), then there exists a unique lower triangular matrix L with positive diagonal entries such that $A = LL^\top$.*

Proof. By induction on n. Write $A = \begin{pmatrix} a_{11} & \mathbf{v}^\top \\ \mathbf{v} & A' \end{pmatrix}$. Since A is SPD, $a_{11} > 0$. Set $\ell_{11} = \sqrt{a_{11}}$, $\mathbf{l} = \mathbf{v}/\ell_{11}$. Then $A' - \mathbf{l}\mathbf{l}^\top$ is SPD of size $(n-1) \times (n-1)$ and we apply the induction hypothesis. □

Algorithm 4: Cholesky factorisation

Input: A SPD, $n \times n$
Output: L such that $A = LL^\top$
for $j = 1, \ldots, n$ **do**
  $\ell_{jj} \leftarrow \sqrt{a_{jj} - \sum_{k=1}^{j-1} \ell_{jk}^2}$;
  for $i = j+1, \ldots, n$ **do**
    $\ell_{ij} \leftarrow \frac{1}{\ell_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik}\ell_{jk} \right)$;

Theorem 3.6 (Cost of Cholesky)**.** $\frac{n^3}{3} + \mathcal{O}(n^2)$*: half the cost of LU.*

3.5 Numerical example

$A = \begin{pmatrix} 4 & 2 \\ 2 & 5 \end{pmatrix}$, $b = \begin{pmatrix} 8 \\ 9 \end{pmatrix}$. Cholesky: $L = \begin{pmatrix} 2 & 0 \\ 1 & 2 \end{pmatrix}$.

Forward substitution $Ly = b$: $y_1 = 4$, $y_2 = (9 - 1 \cdot 4)/2 = 2.5$. Back substitution $L^\top x = y$: $x_2 = 2.5/2 = 1.25$, $x_1 = (4 - 1 \cdot 1.25)/2 = 1.375$.

3.6 Implementation

Python

```python
import numpy as np
from scipy.linalg import lu, cholesky, solve_triangular

# LU
A = np.array([[2., 1, 1], [4, 3, 3], [8, 7, 9]])
b = np.array([4., 10, 24])
P, L, U = lu(A)
y = solve_triangular(L, P @ b, lower=True)
x = solve_triangular(U, y)
print(f"LU: x = {x}")

# Cholesky
A_spd = np.array([[4., 2], [2, 5]])
b_spd = np.array([8., 9])
L_chol = cholesky(A_spd, lower=True)
y = solve_triangular(L_chol, b_spd, lower=True)
x = solve_triangular(L_chol.T, y)
print(f"Cholesky: x = {x}")
print(f"Cout LU (n=100): ~{2*100**3//3:.0f} flops")
```

Julia

```julia
using LinearAlgebra
A = [2.0 1 1; 4 3 3; 8 7 9]
b = [4.0, 10, 24]
F = lu(A)
x = F \ b
println("LU: x = $x")

A_spd = [4.0 2; 2 5]
b_spd = [8.0, 9]
C = cholesky(A_spd)
x = C \ b_spd
println("Cholesky: x = $x")
```

3.7 Exercises

Exercise 3.1 ($\star$). Factorise $A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 8 & 14 \\ 3 & 14 & 34 \end{pmatrix}$ into LU by hand. Verify.

Exercise 3.2 ($\star\star$). Show that if A is SPD, all pivots in Gaussian elimination are positive (no pivoting needed).

Exercise 3.3 ($\star\star$). Compare the computation times of LU vs. Cholesky for $n = 100, 500, 1000, 2000$ on random SPD matrices. Verify the ratio ~ 2.

Exercise 3.4 ($\star\star\star$ – Project)**.** Implement the banded LU factorisation for tridiagonal matrices (Thomas algorithm). Apply to the discretisation of $-u'' = f$.

Key Formulas

$$A = LU \quad (\text{cost } \tfrac{2n^3}{3})$$
$$A = LL^\top \quad (\text{Cholesky, cost } \tfrac{n^3}{3})$$
$$PA = LU \quad (\text{partial pivoting})$$
$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\|, \qquad \frac{\|\delta x\|}{\|x\|} \leq \text{cond}(A) \frac{\|\delta b\|}{\|b\|}$$

Chapter 4

Iterative Methods for Linear Systems

4.1 Motivating example

In finite element methods, the matrix A is sparse ($\mathcal{O}(n)$ nonzero entries out of n^2). LU costs $\mathcal{O}(n^3)$, which is unacceptable for $n = 10^6$. Iterative methods exploit the sparse structure: each iteration costs $\mathcal{O}(n)$.

4.2 General principle

Definition 4.1 (Splitting method)**.** We write $A = M - N$ with M easily invertible. The iteration is:

$$Mx^{(k+1)} = Nx^{(k)} + b, \quad \text{i.e.} \quad x^{(k+1)} = Gx^{(k)} + c,$$

where $G = M^{-1}N$ is the **iteration matrix** and $c = M^{-1}b$.

Theorem 4.2 (Convergence)**.** *The method converges for every $x^{(0)}$ if and only if $\rho(G) < 1$ (spectral radius).*

Proof. $e^{(k)} = x^{(k)} - x^* = G^k e^{(0)}$. Now $\|G^k\| \to 0 \iff \rho(G) < 1$. □

4.3 Jacobi method

Definition 4.3. $A = D - E - F$ (diagonal, lower triangular, upper triangular). $M = D$:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right).$$

Iteration matrix: $G_J = D^{-1}(E + F)$.

4.4 Gauss–Seidel method

Definition 4.4. $M = D - E$:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right).$$

Iteration matrix: $G_{GS} = (D - E)^{-1}F$.

Theorem 4.5 (Convergence for SPD matrices)**.** *If A is SPD, Gauss–Seidel converges.*

Theorem 4.6 (Convergence for diagonally dominant matrices)**.** *If A is strictly diagonally dominant ($|a_{ii}| > \sum_{j \neq i} |a_{ij}|$), then both Jacobi and Gauss–Seidel converge.*

4.5 Successive over-relaxation (SOR)

Definition 4.7. We introduce a parameter $\omega \in (0, 2)$:

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right).$$

$\omega = 1$: Gauss–Seidel. $\omega > 1$: over-relaxation. $\omega < 1$: under-relaxation.

Theorem 4.8. *SOR converges if and only if $0 < \omega < 2$. The optimal ω depends on $\rho(G_J)$.*

4.6 Conjugate gradient method

Definition 4.9. For A SPD, solving $Ax = b$ is equivalent to minimising $\phi(x) = \frac{1}{2}x^\top Ax - b^\top x$.

Algorithm 5: Conjugate gradient

Input: A SPD, b, x_0, tolerance ε
Output: $x \approx A^{-1}b$
$r_0 \leftarrow b - Ax_0$, $p_0 \leftarrow r_0$;
for $k = 0, 1, 2, \ldots$ **do**
  $\alpha_k \leftarrow \frac{r_k^\top r_k}{p_k^\top A p_k}$;
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$;
  $r_{k+1} \leftarrow r_k - \alpha_k A p_k$;
  if $\|r_{k+1}\| < \varepsilon$ **then**
    return x_{k+1}
  $\beta_k \leftarrow \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$;
  $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$;

Theorem 4.10 (Convergence of the conjugate gradient method)**.** *In exact arithmetic, CG converges in at most n iterations. The error satisfies:*

$$\|e_k\|_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k \|e_0\|_A, \quad \kappa = \mathrm{cond}_2(A).$$

4.7 Implementation

Python

```python
import numpy as np

def jacobi(A, b, x0, tol=1e-10, maxiter=1000):
    D = np.diag(np.diag(A))
    R = A - D
    x = x0.copy()
    for k in range(maxiter):
        x_new = np.linalg.solve(D, b - R @ x)
        if np.linalg.norm(x_new - x) < tol:
            return x_new, k+1
        x = x_new
    return x, maxiter

def conjugate_gradient(A, b, x0, tol=1e-10, maxiter=1000):
    x = x0.copy()
    r = b - A @ x
    p = r.copy()
    rs_old = r @ r
    for k in range(maxiter):
        Ap = A @ p
        alpha = rs_old / (p @ Ap)
        x += alpha * p
        r -= alpha * Ap
        rs_new = r @ r
        if np.sqrt(rs_new) < tol:
            return x, k+1
        p = r + (rs_new / rs_old) * p
        rs_old = rs_new
    return x, maxiter

# Test : matrice tridiagonale
n = 100
A = np.diag(2*np.ones(n)) - np.diag(np.ones(n-1), 1) -
↪  np.diag(np.ones(n-1), -1)
b = np.ones(n)
x0 = np.zeros(n)

x_j, nj = jacobi(A, b, x0)
x_cg, ncg = conjugate_gradient(A, b, x0)
print(f"Jacobi: {nj} iterations")
print(f"CG:     {ncg} iterations")
```

Julia

```julia
function cg(A, b, x0; tol=1e-10, maxiter=1000)
    x = copy(x0)
    r = b - A * x
    p = copy(r)
```

```
    rs = dot(r, r)
    for k in 1:maxiter
        Ap = A * p
        alpha = rs / dot(p, Ap)
        x .+= alpha .* p
        r .-= alpha .* Ap
        rs_new = dot(r, r)
        sqrt(rs_new) < tol && return x, k
        p .= r .+ (rs_new / rs) .* p
        rs = rs_new
    end
    return x, maxiter
end
```

4.8 Exercises

Exercise 4.1 ($\star$)**.** Apply 3 iterations of Jacobi and Gauss–Seidel to the system $\begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, $x^{(0)} = (0,0)^\top$.

Exercise 4.2 ($\star\star$)**.** Show that for a tridiagonal matrix T_n of size n, $\rho(G_J) = \cos(\pi/(n+1))$. Deduce the number of Jacobi iterations needed for a precision ε.

Exercise 4.3 ($\star\star\star$ – Project)**.** Compare Jacobi, Gauss–Seidel, SOR and CG for the 2D discretisation of the Laplacian $-\Delta u = f$ on $[0,1]^2$. Plot the number of iterations as a function of n.

Key Formulas

$$\text{Jacobi} : x^{(k+1)} = D^{-1}(b - (A - D)x^{(k)})$$
$$\text{Gauss–Seidel} : G = (D - E)^{-1}F$$
$$\text{CG} : \|e_k\|_A \le 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|e_0\|_A$$

Chapter 5

Polynomial Interpolation

5.1 Motivating example

We have temperature measurements at 5 time instants. We want to estimate the temperature at an intermediate time. Polynomial interpolation constructs a polynomial passing exactly through the measured points.

5.2 The interpolation problem

Theorem 5.1 (Existence and uniqueness)**.** *Given $n+1$ distinct points $(x_0, y_0), \ldots, (x_n, y_n)$, there exists a **unique** polynomial $p_n \in \mathcal{P}_n$ such that $p_n(x_i) = y_i$ for all i.*

Proof. The Vandermonde matrix $V_{ij} = x_i^j$ is invertible since $\det(V) = \prod_{i>j}(x_i - x_j) \neq 0$. □

5.3 Lagrange form

Definition 5.2.

$$p_n(x) = \sum_{i=0}^{n} y_i \, L_i(x), \quad L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

The L_i are the **Lagrange basis polynomials**: $L_i(x_j) = \delta_{ij}$.

5.4 Newton form

Definition 5.3 (Divided differences)**.**

$$f[x_i] = f(x_i), \quad f[x_i, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}.$$

Definition 5.4 (Newton form)**.**

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + \cdots + f[x_0, \ldots, x_n] \prod_{j=0}^{n-1} (x - x_j).$$

Evaluation by the generalised Horner algorithm: $\mathcal{O}(n)$ operations.

5.5 Interpolation error

Theorem 5.5 (Interpolation error). *If $f \in C^{n+1}([a,b])$, then for every $x \in [a,b]$:*

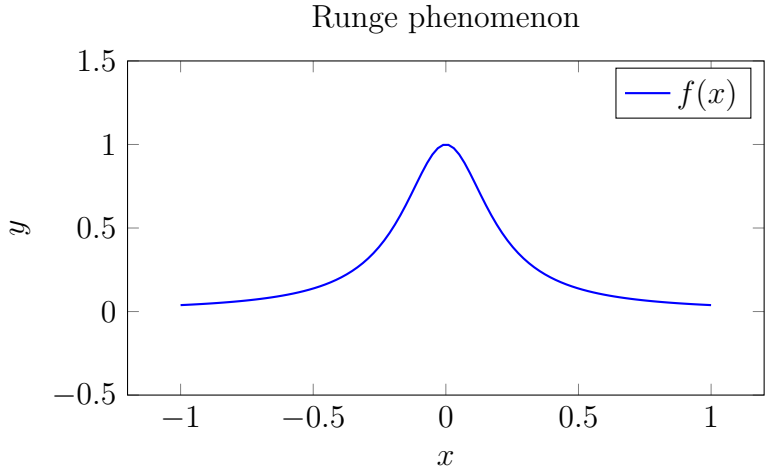
$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!} \prod_{i=0}^{n} (x - x_i),$$

where $\xi_x \in (a,b)$ depends on x.

Proof. Set $w(t) = \prod(t - x_i)$ and $\varphi(t) = f(t) - p_n(t) - \lambda w(t)$ with λ chosen so that $\varphi(x) = 0$ (for a fixed $x \neq x_i$). Then φ vanishes at $n+2$ points. By Rolle's theorem applied $n+1$ times, $\varphi^{(n+1)}$ vanishes at some point ξ. Since $\varphi^{(n+1)} = f^{(n+1)} - \lambda(n+1)!$, we get $\lambda = f^{(n+1)}(\xi)/(n+1)!$. □

Warning

The **Runge phenomenon**: for $f(x) = 1/(1+25x^2)$ on $[-1,1]$ with equidistant nodes, the interpolation error *diverges* as $n \to \infty$ near the endpoints.



5.6 Chebyshev nodes

Definition 5.6. The Chebyshev nodes on $[-1,1]$:

$$x_k = \cos\left(\frac{2k+1}{2(n+1)}\pi\right), \quad k = 0, \ldots, n.$$

They minimise $\max_{x\in[-1,1]} |\prod(x - x_i)|$.

Theorem 5.7. *With Chebyshev nodes:* $\max |w(x)| \leq 2^{-n}$, *which avoids the Runge phenomenon for analytic functions.*

5.7 Cubic splines

Definition 5.8 (Cubic interpolating spline). $s \in C^2([a,b])$, $s|_{[x_i,x_{i+1}]} \in \mathcal{P}_3$, $s(x_i) = y_i$. Boundary conditions:

- **Natural**: $s''(x_0) = s''(x_n) = 0$.
- **Clamped**: $s'(x_0) = f'(x_0)$, $s'(x_n) = f'(x_n)$.

Theorem 5.9 (Cubic spline error). *If $f \in C^4$ and $h = \max(x_{i+1} - x_i)$:*

$$\|f - s\|_\infty \leq \frac{5}{384} h^4 \|f^{(4)}\|_\infty.$$

5.8 Implementation

Python

```python
import numpy as np
from scipy.interpolate import lagrange, CubicSpline
import matplotlib.pyplot as plt

# Interpolation de Lagrange
x_nodes = np.array([-1, -0.5, 0, 0.5, 1])
f = lambda x: 1 / (1 + 25*x**2)
y_nodes = f(x_nodes)
p = lagrange(x_nodes, y_nodes)

x_fine = np.linspace(-1, 1, 300)
plt.figure(figsize=(10, 5))
plt.plot(x_fine, f(x_fine), 'b-', label='f(x)', lw=2)
plt.plot(x_fine, p(x_fine), 'r--', label=f'Lagrange deg
↪ {len(x_nodes)-1}')
plt.plot(x_nodes, y_nodes, 'ko', markersize=8)
plt.legend(); plt.title("Interpolation de Lagrange")
plt.savefig("ch05_lagrange.pdf"); plt.show()

# Noeuds de Tchebychev
n = 15
cheb = np.cos((2*np.arange(n+1)+1)/(2*(n+1))*np.pi)
y_cheb = f(cheb)
p_cheb = lagrange(cheb, y_cheb)
print(f"Erreur equidist (n=15):
↪ {np.max(np.abs(f(x_fine)-lagrange(np.linspace(-1,1,16),
↪ f(np.linspace(-1,1,16)))(x_fine))):.4f}")
print(f"Erreur Tchebychev (n=15):
↪ {np.max(np.abs(f(x_fine)-p_cheb(x_fine))):.6f}")

# Spline cubique
cs = CubicSpline(x_nodes, y_nodes)
print(f"Spline en 0.25: {cs(0.25):.6f}, exact: {f(0.25):.6f}")
```

Julia

```julia
using Interpolations
```

```
f(x) = 1 / (1 + 25x^2)
nodes = [-1.0, -0.5, 0, 0.5, 1.0]
vals = f.(nodes)

# Differences divisees (Newton)
function divided_diff(x, y)
    n = length(x)
    F = copy(y)
    for j in 2:n, i in n:-1:j
        F[i] = (F[i] - F[i-1]) / (x[i] - x[i-j+1])
    end
    return F
end
println("Coefs Newton: ", divided_diff(nodes, vals))
```

5.9 Exercises

Exercise 5.1 ($\star$)**.** Construct the Lagrange polynomial passing through $(0,1)$, $(1,3)$, $(2,7)$ and verify.

Exercise 5.2 ($\star\star$)**.** Prove Theorem 5.5 in detail.

Exercise 5.3 ($\star\star$)**.** Compare the interpolation error with equidistant nodes vs. Chebyshev nodes for $f(x) = 1/(1+25x^2)$, $n = 5, 10, 15, 20$.

Exercise 5.4 ($\star\star\star$ – Project)**.** Implement Hermite interpolation (data for both f and f'). Apply to the construction of Bézier curves.

Key Formulas

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod (x - x_i)$$

$$x_k^{\text{Cheb}} = \cos\left(\frac{2k+1}{2(n+1)}\pi\right)$$

$$\|f - s\|_\infty \leq \frac{5}{384} h^4 \|f^{(4)}\|_\infty$$

Chapter 6

Approximation — Least Squares and Chebyshev

6.1 Introduction

In many practical situations, we have a set of data points and wish to find a function that *best represents* them. Unlike interpolation, which requires the approximating function to pass exactly through every data point, approximation seeks the function that is *closest* in some well-defined sense.

Definition 6.1 (Approximation problem)**.** Given data points (x_i, y_i), $i = 0, 1, \ldots, m$, and a family of functions $\{g(x; a_0, \ldots, a_n)\}$ with $n < m$, the **approximation problem** consists of finding the parameters $a_0, \ldots, a_n$ that minimize a measure of the discrepancy between $g(x_i; a_0, \ldots, a_n)$ and y_i.

The two principal approaches are:

- **Least squares approximation**: minimize the sum of squared residuals;
- **Chebyshev (minimax) approximation**: minimize the maximum absolute error.

6.2 Discrete Least Squares

6.2.1 Problem formulation

Definition 6.2 (Discrete least squares problem)**.** Given $m + 1$ data points (x_i, y_i) for $i = 0, \ldots, m$ and a basis of functions $\varphi_0, \varphi_1, \ldots, \varphi_n$ with $n < m$, we seek coefficients $a_0, a_1, \ldots, a_n$ minimizing

$$S(a_0, \ldots, a_n) = \sum_{i=0}^{m} \left(y_i - \sum_{j=0}^{n} a_j \varphi_j(x_i) \right)^2 .$$

Remark 6.3. The condition $n < m$ is essential: the system is *overdetermined*, so we cannot in general satisfy all equations exactly. When $n = m$, we recover the interpolation problem.

6.2.2 Normal equations

Theorem 6.4 (Normal equations)**.** *The least squares solution satisfies the* ***normal equations****:*

$$\mathbf{A}^\top \mathbf{A}\,\mathbf{a} = \mathbf{A}^\top \mathbf{y},$$

where $\mathbf{A}$ *is the* $(m+1)\times(n+1)$ *matrix with entries* $A_{ij} = \varphi_j(x_i)$*,* $\mathbf{a} = (a_0, \ldots, a_n)^\top$*, and* $\mathbf{y} = (y_0, \ldots, y_m)^\top$*.*

Proof. Define $\mathbf{r} = \mathbf{y} - \mathbf{A}\mathbf{a}$ (the residual vector). Then

$$S(\mathbf{a}) = \|\mathbf{r}\|_2^2 = (\mathbf{y} - \mathbf{A}\mathbf{a})^\top(\mathbf{y} - \mathbf{A}\mathbf{a}) = \mathbf{y}^\top\mathbf{y} - 2\mathbf{a}^\top\mathbf{A}^\top\mathbf{y} + \mathbf{a}^\top\mathbf{A}^\top\mathbf{A}\,\mathbf{a}.$$

Setting the gradient with respect to $\mathbf{a}$ to zero:

$$\nabla_{\mathbf{a}} S = -2\mathbf{A}^\top\mathbf{y} + 2\mathbf{A}^\top\mathbf{A}\,\mathbf{a} = \mathbf{0}.$$

This gives $\mathbf{A}^\top\mathbf{A}\,\mathbf{a} = \mathbf{A}^\top\mathbf{y}$.

Since S is a convex quadratic form (the Hessian $2\mathbf{A}^\top\mathbf{A}$ is positive semi-definite), any critical point is a global minimum. If the columns of $\mathbf{A}$ are linearly independent, then $\mathbf{A}^\top\mathbf{A}$ is positive definite and the solution is unique. □

Example 6.5 (Linear least squares fit)**.** Given the data

x_i	0	1	2	3	4
y_i	1.0	2.1	2.9	4.2	4.8

we fit $g(x) = a_0 + a_1 x$. The matrix $\mathbf{A}$ and the normal equations are:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix}, \qquad \mathbf{A}^\top\mathbf{A} = \begin{pmatrix} 5 & 10 \\ 10 & 30 \end{pmatrix}, \qquad \mathbf{A}^\top\mathbf{y} = \begin{pmatrix} 15.0 \\ 38.8 \end{pmatrix}.$$

Solving: $a_0 = 1.08$, $a_1 = 0.96$, so $g(x) = 1.08 + 0.96x$.

Warning

The matrix $\mathbf{A}^\top\mathbf{A}$ can be very ill-conditioned, especially when using monomials $\varphi_j(x) = x^j$ of high degree. This is why orthogonal polynomials or QR factorization are preferred in practice.

6.2.3 Polynomial least squares via QR factorization

Proposition 6.6. *If* $\mathbf{A} = \mathbf{Q}\mathbf{R}$ *is the (thin) QR factorization of* $\mathbf{A}$ *where* $\mathbf{Q} \in \mathbb{R}^{(m+1)\times(n+1)}$ *has orthonormal columns and* $\mathbf{R} \in \mathbb{R}^{(n+1)\times(n+1)}$ *is upper triangular, then the least squares solution is*

$$\mathbf{R}\,\mathbf{a} = \mathbf{Q}^\top\mathbf{y}.$$

Proof. From $\mathbf{A} = \mathbf{Q}\mathbf{R}$, we get $\mathbf{A}^\top\mathbf{A} = \mathbf{R}^\top\mathbf{Q}^\top\mathbf{Q}\mathbf{R} = \mathbf{R}^\top\mathbf{R}$ and $\mathbf{A}^\top\mathbf{y} = \mathbf{R}^\top\mathbf{Q}^\top\mathbf{y}$. The normal equations become $\mathbf{R}^\top\mathbf{R}\,\mathbf{a} = \mathbf{R}^\top\mathbf{Q}^\top\mathbf{y}$. Since $\mathbf{R}$ is invertible, we may multiply on the left by $(\mathbf{R}^\top)^{-1}$ to obtain $\mathbf{R}\,\mathbf{a} = \mathbf{Q}^\top\mathbf{y}$. □

6.3 Continuous Least Squares and Orthogonal Polynomials

6.3.1 Continuous least squares

Definition 6.7 (Continuous least squares)**.** Given a weight function $w(x) > 0$ on $[a, b]$ and a function $f \in C([a, b])$, we seek a polynomial $p_n \in \mathcal{P}_n$ minimizing

$$\|f - p_n\|_w^2 = \int_a^b w(x)\,[f(x) - p_n(x)]^2\,dx.$$

The solution satisfies the continuous normal equations:

$$\sum_{j=0}^{n} a_j \underbrace{\int_a^b w(x)\,\varphi_j(x)\,\varphi_k(x)\,dx}_{\langle \varphi_j, \varphi_k \rangle_w} = \underbrace{\int_a^b w(x)\,f(x)\,\varphi_k(x)\,dx}_{\langle f, \varphi_k \rangle_w}, \qquad k = 0, \ldots, n.$$

6.3.2 Orthogonal polynomials

Definition 6.8 (Orthogonal polynomials)**.** A sequence $\{p_k\}_{k \geq 0}$ of polynomials (with $\deg p_k = k$) is **orthogonal** with respect to the inner product $\langle \cdot, \cdot \rangle_w$ if

$$\langle p_j, p_k \rangle_w = \int_a^b w(x)\,p_j(x)\,p_k(x)\,dx = 0 \quad \text{for } j \neq k.$$

Theorem 6.9 (Three-term recurrence)**.** *Orthogonal polynomials satisfy a recurrence of the form*

$$p_{k+1}(x) = (\alpha_k x + \beta_k)\,p_k(x) - \gamma_k\,p_{k-1}(x), \qquad k \geq 1,$$

with $p_0(x) = 1$ *and* $p_1(x) = \alpha_0 x + \beta_0$.

Example 6.10 (Classical orthogonal polynomials)**.**

Name	**Interval**	$w(x)$	**Notation**
Legendre	$[-1, 1]$	1	$P_k(x)$
Chebyshev	$[-1, 1]$	$(1 - x^2)^{-1/2}$	$T_k(x)$
Laguerre	$[0, \infty)$	e^{-x}	$L_k(x)$
Hermite	$(-\infty, \infty)$	e^{-x^2}	$H_k(x)$

Proposition 6.11. *If* $\{p_0, p_1, \ldots, p_n\}$ *is an orthogonal basis, the best approximation is*

$$f_n(x) = \sum_{k=0}^{n} c_k\,p_k(x), \qquad c_k = \frac{\langle f, p_k \rangle_w}{\langle p_k, p_k \rangle_w}.$$

The normal equations decouple because the Gram matrix is diagonal.

6.4 Chebyshev Polynomials

Definition 6.12 (Chebyshev polynomials)**.** The Chebyshev polynomials of the first kind are defined by

$$T_k(x) = \cos(k \arccos x), \qquad x \in [-1, 1], \quad k = 0, 1, 2, \ldots$$

They satisfy the recurrence $T_0(x) = 1$, $T_1(x) = x$, $T_{k+1}(x) = 2x\,T_k(x) - T_{k-1}(x)$.

Theorem 6.13 (Properties of Chebyshev polynomials). *1. T_k has exactly k zeros on $(-1,1)$: $x_j = \cos\left(\frac{(2j-1)\pi}{2k}\right)$, $j = 1, \dots, k$.*

2. $|T_k(x)| \le 1$ for all $x \in [-1,1]$.

3. T_k attains the values ± 1 at $k+1$ extremal points: $x_j^* = \cos\left(\frac{j\pi}{k}\right)$, $j = 0, \dots, k$.

4. Orthogonality: $$\int_{-1}^{1} \frac{T_j(x)\,T_k(x)}{\sqrt{1-x^2}}\,dx = \begin{cases} 0 & j \neq k, \\ \pi & j = k = 0, \\ \pi/2 & j = k \neq 0. \end{cases}$$

Theorem 6.14 (Minimax property). *Among all monic polynomials of degree n, the polynomial $\widetilde{T}_n(x) = 2^{1-n} T_n(x)$ has the smallest uniform norm on $[-1,1]$:*

$$\max_{x \in [-1,1]} |\widetilde{T}_n(x)| = 2^{1-n}.$$

6.5 Best Uniform Approximation (Chebyshev / Minimax)

Definition 6.15 (Best uniform approximation). Given $f \in C([a,b])$, the **best uniform (minimax) approximation** of degree n is the polynomial p_n^* satisfying

$$\|f - p_n^*\|_\infty = \min_{p \in \mathcal{P}_n} \|f - p\|_\infty = \min_{p \in \mathcal{P}_n} \max_{x \in [a,b]} |f(x) - p(x)|.$$

Theorem 6.16 (Chebyshev equioscillation theorem). *A polynomial $p_n^* \in \mathcal{P}_n$ is the best uniform approximation to $f \in C([a,b])$ if and only if the error $e(x) = f(x) - p_n^*(x)$ **equioscillates** at least $n+2$ times: there exist $n+2$ points*

$$a \le x_0 < x_1 < \cdots < x_{n+1} \le b$$

such that

$$e(x_j) = (-1)^j\, \sigma\, \|e\|_\infty, \qquad j = 0, 1, \dots, n+1,$$

where $\sigma = \pm 1$.

Remark 6.17. The equioscillation theorem guarantees both existence and uniqueness of the best uniform approximation. The Remez algorithm is a classical iterative method for computing it.

6.6 Near-best Approximation via Chebyshev Interpolation

Theorem 6.18 (Near-optimality of Chebyshev interpolation). *Let p_n^{Cheb} be the interpolation polynomial at Chebyshev nodes and p_n^* the best uniform approximation. Then*

$$\|f - p_n^{\mathrm{Cheb}}\|_\infty \le \left(1 + \Lambda_n^{\mathrm{Cheb}}\right) \|f - p_n^*\|_\infty,$$

where $\Lambda_n^{\mathrm{Cheb}} = \mathcal{O}(\ln n)$ is the Lebesgue constant for Chebyshev nodes.