Math 151A: Applied Numerical Methods

2024-2025
Winter '25

Instructor: Mo Zhou

Textbook: R. Burden & J. Faires - Numerical Analysis

Topics: Floating-point representations, root-finding methods, interpolation, numerical differentiation & integration, methods for solving linear systems

Table of Contents

Calculus Review

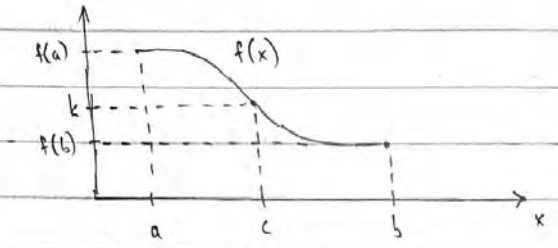
1/6/25

Lecture 1

Calculus Review

"IVT" — continuous fns on $[a,b]$

Theorem (Intermediate Value Theorem): Let $f \in C([a,b])$, and let $k \in \mathbb{R}$ be a real number s.t. $f(a) \le k \le f(b)$ or $f(b) \le k \le f(a)$. Then $\exists$ at least one $c \in [a,b]$ s.t. $f(c) = k$.

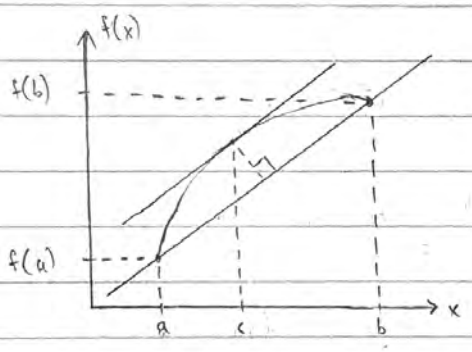


(*) Informally: "continuous" ⇕ "can be drawn w/ one line"

"MVT"

Theorem (Mean Value Theorem): Let $f \in C([a,b])$ be continuous on the closed interval $[a,b]$ and differentiable on the open interval (a,b). Then $\exists$ at least one $c \in (a,b)$ s.t.:

$$\boxed{f'(c) = \frac{f(b) - f(a)}{b - a}}$$



(*) Remarks

(i) The IVT guarantees the existence of solutions to various nonlinear equations

(ii) The MVT is helpful in proving various properties of functions (e.g. monotonicity)

Theorem (Taylor's Theorem)

Let $f \in C^n[a,b]$ be n-times continuously differentiable on the interval $[a,b]$, and let $x_0 \in [a,b]$. Additionally, assume the $(n+1)^{th}$ derivative f^{n+1} exists on $[a,b]$. Then for each $x \in [a,b]$, $\exists\ \zeta_x \in [x_0, x]$ such that:

(1) $x_0 \le \zeta_x \le x$

(2) The function f can be expressed as $f(x) = P_n(x) + R_n(x)$, where: ("n^{th}-degree Taylor polynomial")

$$\boxed{P_n(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n}$$

$$\boxed{R_n(x) = \frac{f^{(n+1)}(x_0)}{(n+1)!}(x - x_0)^{n+1}} \quad \text{"remainder term"}$$

Intro to Floating Point

1/6/25
Lecture 1
+ Lecture 2

(*) Remarks: Taylor's Theorem

- Intuition: If a function f is smooth, then it locally "looks like" a polynomial
- Taylor's theorem approximates f by an n-degree polynomial; remainder term $R_n(x)$ represents the approximation error
  - Remainder term decreases (generally, locally) for larger n
  - $\exists$ various "forms" of the remainder term; as given: Lagrange form

Computational Error

3 big challenges in computation:

(i) Computers have finite memory → can only represent numbers with finite precision

(ii) Only some rational numbers can be represented exactly; irrational numbers, in general, cannot be represented exactly

(iii) Working with fixed-size number representations introduces rounding-off errors + other issues
└ e.g. FP

4 types of error in computation:

(1) Rounding error: Error from representing a number using a finite # of bits

(2) Computational error: Error from performing arithmetic operations using a finite # of bits

(3) Truncation error: Error due to using approximations when computing functions
- Ex: Taylor series for sin & cos; discrete sums for integrals

(4) Error in data: Error from measurement inaccuracies, or representing physical quantities as numbers
└ e.g. data noise

Floating Point Representations

Def: Floating point is the computational form for representing a number (esp. decimal numbers)

(*) Can notate as $FP(x) = x + \varepsilon$, where ε is the rounding error

↓ "FP"

Remark: $\exists$ multiple kinds of floating point representations (corresponding to different levels of precision)

- "Single precision" (AKA "short"): ~6-9 significant digits [base 10] ("float32")
- "Double precision" ("long"): ~15-17 significant digits ("float64")

(*) In general, "machine accuracy" given as $\varepsilon \approx 10^{-16}$; any number with $|x| < \varepsilon$ is simply treated as 0

The Floating Point Representation

1/8/25
Lecture 2 (cont.)

The Floating Point Representation

IEEE binary64 format divided into 3 parts:

1. Sign bit: 1 bit
2. Exponent: 11 bits
3. Significand: 52 bits

→ FP: $(-1)^{\text{sign}} [1.\text{significand}]_2 \times 2^{\text{exponent}-1023}$

Floating Point

- Sign bit determines sign of a number (including zero, which is signed)
- Exponent bits unsigned, range from 0 to 2047 → (exponent - 1023): -1022 to +1023
  - Note: -1023 (all 0s), +1024 (all 1s) reserved for special numbers
- Significand ("mantissa") provides ~15-17 significant decimal digits of precision

Finite-Digit Arithmetic

For numbers that cannot be represented exactly, have to either chop or round

(i) 3-digit chopping: $fl(0.71452\ldots) = 0.714$; $fl(\pi) = 3.14$

(ii) 3-digit rounding: $fl(0.71452\ldots) = 0.715$; $fl(\pi) = 3.14$ ← typically preferred

When doing arithmetic, have to chop/round after every operation:

(*) Ex. (Rounding):

$$(0.714 + 0.333) + 0.462 = (1.047) + 0.462$$

round ↓

$$1.05 + 0.462 = 1.512 \xrightarrow{\text{round}} 1.51$$

true answer: 1.509

Consequence: When doing finite-digit arithmetic, each arithmetic operation introduces / can introduce some amount of error

- Equivalent functions may yield different results depending on number & order of operations

In general, less operations → less error (on average)

→ can use nested arithmetic to minimize # of operations needed by factoring terms

Types of Error

1/8/25
Lecture 2 + Lecture 3

Types of Error

Let p be an approximation of p^* → 2 main types of approximation error:

(1) Absolute error: $|p - p^*|$

(2) Relative error: $\left|\frac{p-p^*}{p^*}\right|$ [for $p^* \neq 0$] ← typically preferred; accounts for magnitude of p^*

Special cases of error:

(i) Underflow error: When a small number is rounded to 0 ← after subtracting 2 nearly-equal #s, e.g. ← try to avoid these!!!

(ii) Overflow error: When a large number is rounded to $\pm\infty$ ← after dividing by a very small #, e.g.

(*) Recall: FP exponent bit all 1s / all 0s reserved for special cases (such as the above)

Big O Notation

Def.: We say that a function $f(x)$ is $O(g(x))$ as $x \to \infty$ if $\exists$ constants $M > 0$, $x_0 \in \mathbb{R}$ s.t.:

$$|f(x)| \leq M|g(x)| \quad \forall\ x > x_0$$

We say that $f(x)$ is $O(g(x))$ as $x \to a$ if $\exists$ constants $M > 0$, $\delta > 0$ s.t.:

$$|f(x)| \leq M|g(x)| \quad \forall\ x \text{ s.t. } |x-a| \leq \delta$$

(*) Intuition: "$f(x) = O(g(x))$" <=> $f(x)$ grows no faster than $g(x)$ asymptotically

Can use big-O notation to describe (e.g.):

(i) Accuracy of an algorithm: Let h a spatial/temporal step size → can say an algorithm's error is $O(f(h_k))$ as $k \to \infty$ to describe asymptotic behavior

(ii) Convergence rate of a sequence: Can use big-O notation to describe the rate at which an approximation converges to a solution

(iii) Computational complexity: Say that an algorithm has a complexity of / is $O(f(n))$ if the number of operations required is $\sim O(f(n))$ for n large

Orders of Convergence

1/13/25
Lecture 3 (cont.)

Convergence of Algorithms (Lec. 3)

Def: A convergent sequence $\{x_n\}_{n\geq 1}$ with limit x is said to converge with order of convergence $\alpha \geq 1$ to a point x if $\exists$ a constant $L \in (0, \infty)$ such that:

$$\lim_{n\to\infty} \frac{|x_{n+1} - x|}{|x_n - x|^\alpha} = L$$

[$L > 0$ called the asymptotic error constant]

Say that a sequence's convergence is:
- linear if $\alpha = 1$ and $L < 1$ ✡
- sub-linear if $\alpha = 1$ and $L = 1$
- super-linear if $\alpha > 1$
- quadratic if $\alpha = 2$ ✡

Root-Finding Algorithms (Lec. 3)

Root-finding: Solving $f(x) = 0$ [i.e. finding x s.t. $f(x) = 0$] for unknown, potentially complicated f'ns f
- Note: Optimization (finding max/min of a function f) often involves root-finding on the derivative f'

General approach: From some initializer x_0, want to construct a sequence $\{x_n\}_{n=1}^{\infty}$ such that x_n converges to a root of $f(x)$.

Typically, stop iterating (generating new x_n's) once some stopping criterion is met, e.g.:
(i) Absolute difference: $|x_n - x_{n-1}| < \varepsilon$ for some tolerance ε
(ii) Relative difference: $|x_n - x_{n-1}| / |x_n| < \varepsilon$ (assuming $x_n \neq 0$)
(iii) Residual: $|f(x_n)| < \varepsilon$

(*) Q- and R-Convergence

Often want to talk about the convergence rate of root-finding algorithms (i.e. of the sequences generated by the algorithm, and that converge to x s.t. $f(x) = 0$) → distinguish between Q- and R-convergence:
1. An algorithm has Q-convergence [with rate α] if all generated sequences converge with exactly order α
2. An algorithm has R-convergence [with rate α] if all generated sequences converge with order at least α

The Bisection Method

1/15/25

Lecture 4

Bisection Methods (Lec. 4)

Recall (IVT): f continuous on $[a,b]$, $\text{sign}[f(a)] \neq \text{sign}[f(b)] \Rightarrow \exists\ p \in [a,b]$ s.t. $f(p) = 0$

→ **Idea:** Keep shrinking the interval $[a,b]$ to obtain increasingly accurate approximations of p

Bisection Method Algorithm

1. Initialize $a_1 = a$, $b_1 = b$, and let $p_1 = (a_1 + b_1)/2$
2. while [stopping criterion] is not satisfied by p_n:
   (a) if $f(p_n) \cdot f(a_n) > 0 \Rightarrow$ then $p \in [p_n, b_n] \rightarrow \underline{a_{n+1} = p_n,\ b_{n+1} = b_n}$
   if $f(p_n) \cdot f(a_n) < 0 \Rightarrow$ then $p \in [a_n, p_n] \rightarrow \underline{a_{n+1} = a_n,\ b_{n+1} = p_n}$
   (b) take $p_{n+1} = (a_{n+1} + b_{n+1})/2$

   [repeat until stopping criteria is met]

Remarks:

- If there are multiple roots $p' \in [a,b]$, the bisection method will find 1 of them [no guarantee which]
- The bisection method converges globally: as long as f is continuous on $[a,b]$ and $\text{sign}[f(a)] \neq \text{sign}[f(b)]$, the bisection method will converge to a root

$$\left(\text{sign}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}\right.$$

Convergence of the Bisection Method (Lec. 4)

Thm: Let $f \in C[a,b]$ s.t. $\text{sign}(f(a)) \neq \text{sign}(f(b))$. Then the bisection method generates a sequence $\{p_n\}_{n \geq 1}$ that converges to a root p of f with error bound:

$$\boxed{|p_n - p| \leq \frac{b-a}{2^n} \quad \forall\ n \geq 1}$$

(*) **Proof:** Let $p \in [a,b]$ the root; have that:

$$|p_1 - p| \leq \frac{1}{2}(b-a) \longrightarrow |p_2 - p| \leq \frac{1}{2}(b_2 - a_2) = \frac{1}{2^2}(b-a) \xrightarrow{\text{(via induction)}} \cdots \rightarrow |p_n - p| \leq \frac{1}{2^n}(b-a). \quad \blacksquare$$

Corollary: The bisection method converges linearly (i.e. with order $\alpha = 1$).

Fixed-Point Iteration

1/15/25
Lecture 4
+Lecture 5

The Bisection Method (Pros & Cons)

Pros: Easy to implement; globally convergent; cheap to evaluate (only 1 function call / iteration)

Cons: Slow convergence (linear); cannot find roots of even multiplicity (e.g. in $f(x) = x^2$); cannot be easily generalized to higher dimensions

Fixed-Point Iteration Methods (Lec. 5)

Def: Fixed Point

A fixed point of a function g is a point p in the domain of g satisfying $g(p) = p$.

→ Notice: p a fixed point of g $\iff$ p a root of $f(x) = x - g(x)$.

Thm. (Existence of Fixed Points): Let $g \in C[a, b]$ with $a \le g(x) \le b \ \forall \ x \in [a, b]$; then $\exists$ at least one fixed point p such that $g(p) = p$.

- Proof: Either a or b a fixed point, or via IVT & continuity of $f(x) = x - g(x)$.

Thm. (Uniqueness of Fixed Points): If $g'(x)$ exists on (a, b) and $\exists$ a constant $0 < k < 1$ s.t. $|g'(x)| \le k \ \forall \ x \in (a, b)$, then the aforementioned fixed point is unique.

- Proof: Let $p_1 > p_2$ two fixed points (for contradiction)

→ by MVT: $\exists \ \xi \in (p_1, p_2)$ s.t. $g'(\xi) = \dfrac{g(p_1) - g(p_2)}{p_1 - p_2} = \dfrac{p_1 - p_2}{p_1 - p_2} = 1 > k$ (contradiction) ▨

Fixed-Point Iteration (Lec. 5)

Given $g \in C[a, b]$ s.t. $g(x) \in [a, b] \ \forall \ x \in [a, b]$, want to find a sequence p_n converging to a fixed point p:

1. Choose an initial guess $p_0 \in [a, b]$
2. Generate a sequence $\{p_n\}_{n \ge 1}^{\infty}$, where $p_n = g(p_{n-1})$
   → If the algorithm converges to some $p \in [a, b]$, then p is a fixed point of g

Thm. (Fixed-Point Theorem): Let $g \in C[a, b]$ be s.t. $g(x) \in [a, b] \ \forall \ x \in [a, b]$. Suppose, also, that g' exists on (a, b) and that $\exists$ a constant $k \in (0, 1)$ s.t. $|g'(x)| \le k \ \forall \ x \in (a, b)$.

Then for any $p_0 \in [a, b]$, the sequence defined by $p_n = g(p_{n-1})$ $[n \ge 1]$ converges to the unique fixed point $p \in [a, b]$ of g.

1/17/25
Lecture 5 (cont.)

Fixed-Point Iteration (Lec. 5, cont.)

Proof (Fixed-Point Thm.): Know that $\exists$ unique fixed point $p \in [a,b]$. By the MVT, $\exists\ \xi_n \in [p_n, p]$ s.t.:

$$|p_n - p| = |g(p_{n-1}) - g(p)| = |g'(\xi_n)(p_{n-1} - p)| \leq k|p_{n-1} - p|$$

$$\rightarrow \text{(by induction)} \leq k^n |p_0 - p| \xrightarrow[n\to\infty]{} 0 \quad [0 < k < 1].$$

Corollary: If g satisfies the hypotheses of the Fixed-Point Theorem, then the error $|p_n - p|$ satisfies:

$$\boxed{|p_n - p| \leq k^n \cdot \max\{p_0 - a, b - p_0\}}$$

Thm. (Convergence of Fixed-Point Iteration): The sequence $p_n = g(p_{n-1})$ generated by fixed-point iteration converges linearly to the fixed point p.

- Proof:

By Taylor's theorem, $\exists\ \xi_x$ between $x \in (a,b)$ and p s.t.:

$$g(x) = g(p) + g'(\xi_x)(x - p)$$

Taking $x = p_{n-1}$ and $x_0 = p$, find that:

$$p_n = g(p_{n-1}) = g(p) + g'(\xi_n)(p_{n-1} - p) \quad [\xi_n \text{ between } p_{n-1}, p]$$

$$\rightarrow \frac{p_n - p}{p_{n-1} - p} = g'(\xi_n)$$

Since g' is continuous and $\lim_{n\to\infty} |\xi_n - p| = 0$, have $\lim_{n\to\infty} g'(\xi_n) = g'(p)$

$$\rightarrow \lim_{n\to\infty} \frac{|p_n - p|}{|p_{n-1} - p|} = |g'(p)| \leq k < 1$$

Newton's Method

1/20/25

Lecture 6

Newton's Method (Lec. 6)

Motivation: Bisection method converges linearly; want a method that converges faster (e.g. quadratic)

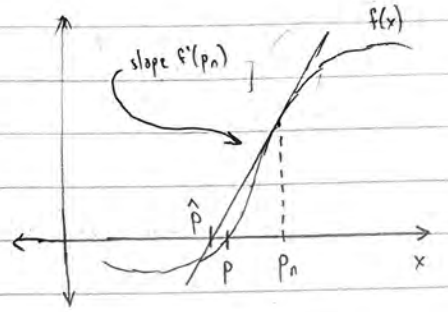
Background:

Let $f \in C^2([a,b])$ and $f(p)=0$; then for p_n near p:

$$f(p) = f(p_n) + f'(p_n)(p_n - p) + \frac{1}{2}f''(\xi)(p-p_n)^2$$

for some ξ between p_n, p. Since $f(p)=0$, ignoring the $()^2$ term:

$$p \approx p_n - \frac{f(p_n)}{f'(p_n)}$$



Newton's Method for Root-Finding

1. Start with an initial guess p_0 s.t. $|p_0 - p|$ is small
2. Until some stopping criterion, generate $\{p_n\}_{n=1}^{\infty}$ by:

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)}$$

Remarks: (i) The Taylor expansion is a local approximation of f (in our case: a tangent line to f), hence the need for $|p_n - p|$ small [at $(p_n, f(p_n))$]

(ii) Newton's method is a <u>local</u> method → may fail to converge for p_0 far from p

Convergence of Newton's Method (Lec. 6)

Thm: Let $f \in C^2([a,b])$. If $p \in (a,b)$ is such that $f(p)=0$ and $f'(p) \neq 0$, then $\exists$ a radius of convergence $\delta > 0$ s.t. Newton's method generates a sequence $\{p_n\}_{n=1}^{\infty}$ converging to p for any initial guess $p_0 \in [p-\delta, p+\delta]$.

Proof: Can consider the fixed-point map

$$g(x) = x - \frac{f(x)}{f'(x)}$$

→ can write Newton's method as $p_{n+1} = g(p_n)$.

1/20/25
Lecture 6 (cont.)

Convergence of Newton's Method (Lec. 6)

Proof (cont.):

Goal: Want to find an interval $[p-\delta, p+\delta]$ s.t. $g(x) \in [p-\delta, p+\delta]$ and:

$$|g'(x)| \leq k < 1 \quad \forall \; x \in (p-\delta, p+\delta)$$

→ can use fixed-point theorem (seen previously).

Since f is continuous and $f'(p) \neq 0$, $\exists\ \delta_1 > 0$ s.t. $f'(x) \neq 0$ for $[p-\delta_1, p+\delta_1] \subseteq [a, b]$.
Hence g is well-defined and continuous on $[p-\delta_1, p+\delta_1]$

For $x \in [p-\delta_1, p+\delta_1]$, have that:

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

$$[f(p) = 0] \rightarrow g'(p) = \frac{f(p)f''(p)}{f'(p)^2} = 0$$

Since g is continuous and $g'(p) = 0$, $\exists\ 0 < \delta \leq \delta_1$ and $0 < k < 1$ s.t.:

$$|g'(x)| \leq k \quad \forall \; x \in [p-\delta, p+\delta]$$

By the MVT, for $x \in [p-\delta, p+\delta]$, $\exists\ \xi$ between x, p s.t.:

$$|g(x) - g(p)| = |g'(\xi)| \cdot |x - p|$$

$$\rightarrow |g(x) - p| = |g(x) - g(p)| = |g'(\xi)| \cdot |x-p| \leq k|x-p| < |x-p| < \delta$$

Hence $g(x) \in [p-\delta, p+\delta]$.
Then by the Fixed-Point Theorem, the sequence $\{p_n\}_{n=1}^{\infty}$ defined by

$$p_n = g(p_{n-1}) = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})} \quad [n \geq 1]$$

converges to p for any $p_0 \in [p-\delta, p+\delta]$.

1/22/25
Lecture 7

Convergence of Newton's Method (cont.)

Convergence Order of Newton's Method (cont.)

Thm: Let $g \in C^\alpha([a,b])$ for $\alpha > 2$ [integer], and let $p \in [a,b]$. Assume:

(i) $g(p) = p$

(ii) $g'(p) = g''(p) = \dots = g^{\alpha-1}(p) = 0$, but $g^\alpha(p) \neq 0$

Let $\{p_n\}_{n=1}^\infty$ defined by $p_{n+1} = g(p_n)$, and assume $\lim_{n\to\infty} p_n = p$. Then p_n will converge to p for all p_0 close to p with order of convergence α.

Corollary: Newton's method has at-least quadratic order of convergence when $f'(p) = 0$.

(i) If $g''(p) \neq 0$, the order of convergence is 2. ≻ locally

(ii) If $g''(p) = 0$, the order of convergence is greater than 2.

Proof: By Taylor's theorem (centered at p):

$$g(p_n) = g(p) + (p_n - p)g'(p) + \dots + \frac{(p_n-p)^{\alpha-1}}{(\alpha-1)!} g^{\alpha-1}(p) + g^\alpha(\xi_n)\frac{(p_n-p)^\alpha}{\alpha!}$$

(ξ_n between p_n, p)

By the theorem conditions on g:

$$g(p_n) = g(p) + g^\alpha(\xi_n)\frac{(p_n-p)^\alpha}{\alpha!}$$

$$[p_{n+1} = g(p_n)] \to p_{n+1} = p + g^\alpha(\xi_n)\frac{(p_n-p)^\alpha}{\alpha!}$$

$$\to \frac{|p_{n+1}-p|}{|p_n-p|^\alpha} = \frac{1}{\alpha!}\cdot|g^\alpha(\xi_n)|$$

Taking the limit, by continuity of g^α and convergence of $\{p_n\}$:

$$\lim_{n\to\infty}\frac{|p_{n+1}-p|}{|p_n-p|^\alpha} = \lim_{n\to\infty}\frac{1}{\alpha!}|g^\alpha(\xi_n)| = \frac{1}{\alpha!}|g^\alpha(p)| > 0$$

Remarks: (i) In practice: often use bisection method to find a good initial guess p_0

(ii) Can look at how $|f'(p_n)|$ affects step size

- $f'(p_n) = 0$ → method fails (unless $f(p_n) = 0$, i.e. p_n is the root)
- $|f'(p_n)| \gg 1$ → updates between iterates are small
- $|f'(p_n)| \ll 1$ → large updates
  ↳ may impede convergence

The Secant Method

1/22/25

The Secant Method (Lec. 7)

Lecture 7 (cont.)

Motivation: Although Newton's method is fast, computing the derivative f' may be computationally expensive; instead, can try to approximate $f'(p_n)$ without computing it explicitly

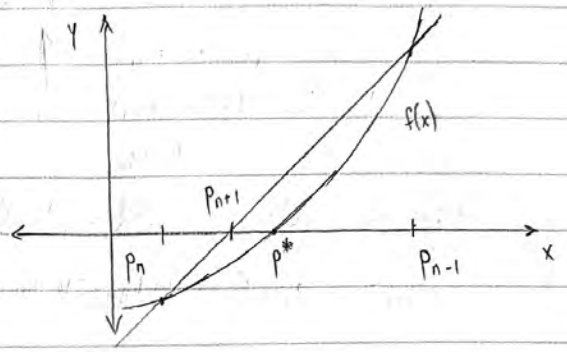
Background:

By definition of the derivative:

$$f'(p_{n-1}) = \lim_{x \to p_{n-1}} \frac{f(x) - f(p_{n-1})}{x - p_{n-1}}$$

For p_{n-2} near p_{n-1}, we can approximate this by:

$$f'(p_{n-2}) \approx \frac{f(p_{n-1}) - f(p_{n-2})}{p_{n-1} - p_{n-2}}$$



The Secant Method for Root-Finding

1. Start with initial guesses p_0, p_1 s.t. $|p_0 - p|, |p_1 - p|$ small
2. Until some stopping criterion, generate $\{p_n\}_{n \geq 1}$ by:

$$p_{n+1} = p_n - \frac{f(p_n)(p_n - p_{n-1})}{f(p_n) - f(p_{n-1})}$$

Root-Finding Algorithms (Lec. 7)

Method	Order of Convergence
Bisection	1
Fixed-Point	1 if $\lvert g'(p)\rvert \in (0,1)$, ≥ 2 if $g'(p) = 0$
Newton's	≥ 2 if $g'(p) \neq 0$; 1 oth.
Secant	$(1+\sqrt{5})/2 \approx 1.618$

← (*) Usually = 2

Notes:

- The bisection method converges globally; others locally
- Newton's method is 1st-order [requires 1st derivative]; often a good choice within its ROC (Other methods are 0th-order, generally slower)

1/22/25

Lecture 7 (cont.)

Convergence of Secant Method

Rate of convergence for the secant method: $(1+\sqrt{5})/2$

(*) Proof:

Denote error $e_n = p_n - p$, and assume p_n is close to p:

Claim: this is approximately a constant C

$$e_{n+1} = e_n - \frac{f(p_n)(e_n - e_{n-1})}{f(p_n) - f(p_{n-1})} = \frac{f(p_n)e_{n-1} - f(p_{n-1})e_n}{f(p_n) - f(p_{n-1})} = e_{n-1}e_n \cdot \frac{1}{f(p_n) - f(p_{n-1})}\left(\frac{f(p_n)}{e_n} - \frac{f(p_{n-1})}{e_{n-1}}\right)$$

Assuming the claim: $\lim_n |e_{n+1}|/|e_n|^\alpha = K \Rightarrow K|e_n|^\alpha = K^{\alpha+1}|e_{n-1}|^{\alpha^2}$ ← α: order of convergence

$$e_{n+1} = e_{n-1}e_n \cdot C \longrightarrow |e_{n+1}| \approx L|e_n|^2 \approx C|e_{n-1}||e_n| = C|e_{n-1}| \cdot L|e_{n-1}|^\alpha = (CL)|e_{n-1}|^{\alpha+1}$$

(CL: some constant)

Have: $\alpha^2 = \alpha + 1 \Rightarrow \alpha = \frac{1}{2}(1+\sqrt{5})$. [faster than Bisection/slower than Newton]

Remains to show that the "constant" is a constant (approx.):

$$C := \frac{1}{f(p_n) - f(p_{n-1})}\left(\frac{f(p_n)}{e_n} - \frac{f(p_{n-1})}{e_{n-1}}\right) = \frac{p_n - p_{n-1}}{f(p_n) - f(p_{n-1})} \cdot \frac{1}{p_n - p_{n-1}}\left(\frac{f(p_n) - f(p)}{p_n - p} - \frac{f(p_{n-1}) - f(p)}{p_{n-1} - p}\right)$$

By Taylor expansion on $f(p_n)$, $f(p_{n-1})$, $f(p)$, have that: ← "forward difference formula" (see later)

$$\frac{f(p_n) - f(p)}{p_n - p} = f'\left(\frac{p_n + p}{2}\right) + O(e_n^2), \quad \frac{f(p_{n-1}) - f(p)}{2} = f'\left(\frac{p_{n-1} + p}{2}\right) + O(e_{n-1}^2)$$

+ by IVT, have:

$$\frac{p_n - p_{n-1}}{f(p_n) - f(p_{n-1})} = \frac{1}{f'(\xi)} \text{ for some } \xi \in [p_{n-1}, p_n]; \approx 1/f'(p) \text{ [if } f'(p) \neq 0]$$

→ Obtain our result:

$$C = \frac{1}{2f'(\xi)} \cdot \frac{2}{p_n - p_{n-1}}\left(f'\left(\frac{p_n + p}{2}\right) + O(e_n^2) - f'\left(\frac{p_{n-1} + p}{2}\right) - O(e_{n-1}^2)\right)$$

$$\approx \frac{1}{2f'(p)}\left[f''\left(\frac{p_n + p_{n-1} + 2p}{4}\right) + O(e_{n-1}^2) + O(e_{n-1})\right] \approx \frac{f''(p)}{2f'(p)}$$ } constant

∎

Newton's Method w/ Multiple Roots

1/24/25

Lecture 8

Newton's Method with Multiple Roots (Lec. 8)

Recall: Newton's method has rate of convergence: (i) $f' \neq 0 \rightarrow$ at-least quadratic [good]

(ii) $f' = 0 \rightarrow$ linear ← want to speed this up

In particular: when $f'(p) = 0$, then for p_n near p: $1/f'(p_n)$ large $\Rightarrow$ slow convergence; want to fix this

Def: A solution p for $f(p) = 0$ is a zero of multiplicity m of f if, for $x \neq p$, can write:

$$f(x) = (x-p)^m q(x) \text{ for } q(x) \text{ s.t. } q(p) \neq 0$$

(This course: only consider integer multiplicities)

↓

Thm: Let $f \in C^m(a,b)$ and $p \in (a,b)$. Then p is a zero [of f] with multiplicity m iff:

$$0 = f(p) = f'(p) = \dots = f^{m-1}(p), \text{ and } f^m(p) \neq 0$$

+

Corollary: Let $m \geq 1$, and let $p \in (a,b)$ a zero of $f \in C^m(a,b)$ with multiplicity m.

Then the function $\mu(x)$ defined by:

$$\mu(x) := \frac{f(p)}{f'(p)}$$

has a zero of multiplicity 1 at point p.

↳ (*) Proof of Corollary

$$f(x) = (x-p)^m q(x) \Rightarrow f'(x) = m(x-p)^{m-1} q(x) + (x-p)^m q(x) \rightarrow$$

$$\rightarrow \mu(x) = \frac{f}{f'} = \frac{(x-p)^m q(x)}{m(x-p)^{m-1} q(x) + (x-p)^m q(x)} = \frac{(x-p) q(x)}{m q(x) + (x-p) q(x)} \Rightarrow \mu(p) = 0, \ \mu'(p) \neq 0 \ \blacksquare$$

Modified Newton's Method

Let $f \in C^m(a,b)$, and p a root of f w/ multiplicity m. Given initial guess p_0 near p, generate $\{p_n\}_{n \geq 1}$ by:

$$p_{n+1} = p_n - \frac{\mu(p_n)}{\mu'(p_n)} = p_n - \frac{f(p_n) f'(p_n)}{[f'(p_n)]^2 - f(p_n) f''(p_n)}$$

Aitken's Δ^2 Method

1/24/25
Lecture 8 + Lecture 9

Remarks (Modified Newton's Method)

- In theory, the method converges quadratically (from quadratic convergence of Newton's method); in practice, may encounter round-off errors in the denominator term
- A drawback: method is 2nd-order (requires computing f'') → may be computationally expensive

Accelerating Convergence (Lec. 8)

Motivation: Given a sequence $\{p_n\}_{n\geq 1}$ that converges to a limit p linearly, want to use $\{p_n\}_{n\geq 1}$ to construct another sequence $\{\hat{p}_n\}_{n\geq 1}$ that converges to p faster

Recall: Let $p_n \to p$ (exactly) linearly; then $e_n = |p_n - p|$ is (roughly) a geometric sequence, i.e. $\lim_{n\to\infty} |p_n - p| / |p_{n-1} - p| = L < 1$.

Theorem (Aitken's Δ^2 Method): Assume $\{p_n\}_{n\geq 1}$ converges linearly to p, and that for n large, $(p_{n+2} - p)(p_n - p) > 0$. Then the sequence $\{\hat{p}_n\}_{n\geq 1}$ given by:

$$\hat{p}_n = p_n - \frac{(p_{n+1} - p_n)^2}{p_{n+2} - 2p_{n+1} + p_n} \quad \text{for } n \geq 0$$

satisfies $\lim_{n\to\infty} |\hat{p}_n - p| / |p_n - p| = 0$.

→ **Approach**: Use p_n to find (faster-converging) $\hat{p}_n$: $p_0, p_1, p_2, p_3, p_4, \ldots$ → $\hat{p}_0, \hat{p}_1, \hat{p}_2 \ldots$

(*) Wrap-Up: Newton's Method (Lec. 9)

Claim: Newton's method converges linearly (locally) with $L = 1 - 1/m$ to roots p with multiplicity $m > 0$.

Proof:

Let $f(x) = (x-p)^m q(x)$ → define $g(x) = x - f/f' = x - \frac{(x-p)^m q}{m(x-p)^{m-1}q + (x-p)^m q'} = x - \frac{(x-p)q(x)}{mq(x) + (x-p)q'(x)}$.

→ **Recall (Fixed-Point)**: $|g'(x)| \leq k < 1 \ \forall x \Rightarrow$ method converges linearly

Here: $g' = 1 - \frac{(q + (x-p)q')(mq + (x-p)q') - (x-p)q(mq + (x-p)q')}{(mq + (x-p)q')^2}$

→ $g'(p) = 1 - q \cdot mq / (mq)^2 = 1 - 1/m$ [constant, < 1]

→ $\lim_{n\to\infty} |p_{n+1} - p| / |p_n - p| = 1 - 1/m$.

∎

Lagrange Polynomials

1/27/25
Lecture 9 (cont.)

Data Fitting (Lec. 9)

Idea: Given data points (x,y) assumed to be taken from some unknown function f, want to (from the points (x,y)) find an approximation $\hat{f}$ for f.

- $\exists$ various uses/applications for data fitting in theory & in practice
- 3 criteria: approximation accuracy, efficiency (i.e. ability to approximate well with relatively little data), generalizability [ability to use similar $\hat{f}$ across various problems]

Thm. (Weierstrass Approximation Theorem):
Let $f \in C^n([a,b])$. Then for any $\varepsilon > 0$, $\exists$ polynomial $P(x)$ such that:

$$|f(x) - P(x)| < \varepsilon \quad \forall\, x \in [a,b]$$

↳ Consequently: given any continuous f, can approximate f to arbitrary accuracy using polynomials.

- Note: This is not always the Taylor expansion!!!
  (Intuitively: Taylor expansion a local approx. → only uses one point; fails for $[a,b]$ large)

Lagrange Polynomials (Lec. 9)

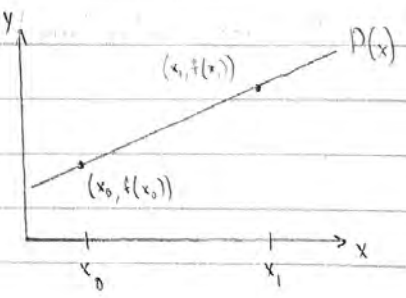
Goal: Given a set of discrete points $\{(x_n, f(x_n))\}$ $[x_i \neq x_j \ \forall\, i \neq j]$, want to find a polynomial $P(x)$ s.t. $P(x_i) = f(x_i)$ $\forall\, i = 1, \dots, n$.

Looking at $n=2$: have 2 points $(x_0, f(x_0))$, $(x_1, f(x_1))$

$$\rightarrow P(x) = \frac{f(x_1)(x-x_0) - f(x_0)(x-x_1)}{x_1 - x_0} = \underbrace{\frac{x-x_1}{x_0-x_1}}_{L_0(x)} f(x_0) + \underbrace{\frac{x-x_0}{x_1-x_0}}_{L_1(x)} f(x_1)$$

(lines)

Notice: (i) $L_0(x)$, $L_1(x)$ form a basis for vector space of 1D polynomials
(ii) $L_0(x_0) = 1$, $L_0(x_1) = 0$ [& vice versa for L_1]



→ $f(x_0)$, $f(x_1)$ are the basis coefficients for $L_0(x)$, $L_1(x)$ in $P(x)$:

$$P(x) = L_0(x) f(x_0) + L_1(x) f(x_1)$$

Lagrange Polynomials (cont.)

1/27/25
Lecture 9
+Lecture 10

Lagrange Polynomials (cont.)

Can generalize notion of basis functions to an arbitrary # [n] of points.

Notice: $L_{k,n}(x_k) = 1$, $L_{k,n}(x_{i\neq k}) = 0$

Define:

$$L_{k,n} := \frac{(x-x_0)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_n)}{(x_k-x_0)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_n)} = \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{x-x_i}{x_k-x_i} \quad (\text{for } k=0,\dots,n)$$

$$\rightarrow \quad P(x) = \sum_{k=0}^{n} f(x_k) L_{k,n}(x)$$

← call this a [the] Lagrange polynomial [degree n]

↳ notice: (n+1) data points → polynom. deg. n

Remark: The set $\{L_{k,n}\}_{0\leq k\leq n}$ forms a basis for the vector space of degree-n polynomials

→ **Corollary**: The coefficients $f(x_0), \dots, f(x_n)$ define the unique polynomial $P(x)$

(*) **Proof of Corollary**

Let $P, Q \in \mathcal{P}^n$ [vector space of polynomials degree $\leq n$] s.t. $P(x_k) = Q(x_k) = f(x_k)$ $\forall\, k = 0, \dots, n$.

Know: $P-Q \in \mathcal{P}^n$, $[P-Q](x_k) = 0$ for $k = 0, \dots, n$

$(P-Q)$ is of degree at most n, but has (n+1) distinct roots $[x_0, \dots, x_n] \Rightarrow P-Q = 0$ [0 polynomial]. ▨

Thm. (Error for Lagrange Interpolation): Let $x_0, \dots, x_n \in [a,b]$ distinct, and let $f \in C^{n+1}[a,b]$. Then for each $x \in [a,b]$, $\exists\, \xi(x) \in [a,b]$ within the interval spanned by $x_0, \dots, x_n$ such that:

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

($P(x)$ the Lagrange polynomial)

(*) **Proof**: Via Rolle's theorem (n times) on $g(x) := f(x) - P(x)$

Can use theorem to derive an upper bound on $|f(x) - P(x)|$

(*) **Notice**: Similar form to the Taylor remainder

Neville's Method

1/31/25
Lecture 11

Neville's Method (Lec. 9)

Motivation: Given a Lagrange polynomial fitted to some existing data: assuming we receive some new data we want to fit to, want to be able to reuse & update our polynomial without having to recompute it.

Notation: Let f be defined at points $\{x_i : 0 \le i \le n\}$, and let $m_1, \dots, m_k \in \{0, 1, \dots, n\}$ distinct. Then we write $P_{m_1 m_2 \dots m_k}(x)$ to denote the Lagrangian polynomial from interpolating $f(x)$ at points $\{x_{m_1}, \dots, x_{m_k}\}$.

Thm: Let f be defined at $x_0, \dots, x_k$, and let $x_i \neq x_j$ be two such points. Then the Lagrangian polynomial that interpolates f at all $(k+1)$ points is given by:

$$P(x) = \frac{(x - x_j) P_{012\dots(j-1)(j+1)\dots k}(x) - (x - x_i) P_{012\dots(i-1)(i+1)\dots k}(x)}{x_i - x_j}$$

(*) $P_i(x) = f(x_i)$

(*) **Ex** ($k=2$):

$$P_{01}(x) = \frac{(x - x_0) P_1 - (x - x_1) P_0}{x_1 - x_0} \qquad P_{12}(x) = \frac{(x - x_1) P_2 - (x - x_2) P_1}{x_2 - x_1}$$

$$\to P_{012}(x) := \sum_{i=0}^{k} P_i L_{k,i}(x) = \frac{(x - x_0) P_{12}(x) - (x - x_2) P_{01}(x)}{x_2 - x_0} = f(x_i) \text{ for } i = 0, 1, 2$$

[Recall: $P_1, P_2 \in P^n$ and $P_1(x_i) = P_2(x_i)$ for $i = 0, \dots, n \Rightarrow P_1(x) = P_2(x)\ \forall x$]

Notice: Given 0-degree polynomials $P_0, \dots, P_k$ $[P_i = f(x_i)]$, can iteratively generate higher-order polynomial approximations of any point x:

$$\begin{array}{ll} x_0 & P_0 \searrow \\ x_1 & P_1 \rightarrow P_{0,1} \searrow \\ x_2 & P_2 \rightarrow P_{1,2} \rightarrow P_{0,1,2} \searrow \\ x_3 & P_3 \rightarrow P_{2,3} \rightarrow P_{1,2,3} \rightarrow P_{0,1,2,3} \end{array}$$

← can fill in left → right, top → bottom

Notation: For $0 \le j \le i$, denote / define: $Q_{i,j} := P_{i-j, i-j+1, \dots, i}$. (*) **Note:** $Q_{i,i} = P_{0,1,\dots,i}$

Remarks: (i) We need not store the entire polynomials $Q_{i,j}$, only their values for a particular point x
(ii) To approximate $P(x)$ [at some x], can stop early when $|Q_{i,i} - Q_{i+1,i+1}| <$ * some tolerance *.

The Divided Differences Method

1/31/25
Lecture 11
+Lecture 12

Neville's Algorithm

Given points $x_0, \dots, x_n$ and values $Q_{0,0}, Q_{1,0}, \dots, Q_{n,0}$ $[Q_{i,0} = f(x_i)]$:

→ For $i = 1, 2, \dots, n$:

For $j = 1, 2, \dots, i$:

Compute:

$$Q_{i,j} = \frac{(x - x_{i-j})Q_{i,j-1} - (x - x_i)Q_{i-1,j-1}}{x_i - x_{i-j}}$$

The Divided Differences Method (Lec. 12)

Def: Let $x_0, \dots, x_n$ be distinct, and let $P(x)$ be the Lagrange polynomial of $\{(x_i, f(x_i))\}$.

Newton's divided differences is a way to write $P(x)$; namely, in the following form:

$$P_n(x) := a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \dots + a_n(x - x_0)\dots(x - x_{n-1})$$

(*) Motivation: Can use the divided difference method to recursively generate higher-degree polynomials

Observations: (i) $P_n(x_0) = a_0$

(ii) $P_n(x_1) = a_0 + a_1(x - x_0) = f(x_1) \Rightarrow a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$

(iii) In general: when computing $P_n(x_k)$, first $(k+1)$ terms will be nonzero (except for $a_i = 0$)

Notation: Denote divided differences by:

$a_0 = f[x_0]$

0^{th} divided difference: $f[x_i] = f(x_i)$

$a_1 = f[x_0, x_1]$

1^{st} divided difference: $f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}$

$\vdots$

k^{th} divided difference:

$$f[x_i, x_{i+1}, \dots, x_{i+k}] = \frac{f[x_{i+1}, \dots, i+k] - f[x_i, \sim, x_{i+k-1}]}{x_{i+k} - x_i}$$

The Divided Differences Method (cont.)

2/3/25
Lecture 12 (cont.)

Remark: The k^{th} divided difference can be computed recursively using the $(k-1)^{th}$ divided difference

Def: Newton's Divided Difference Interpolating Polynomial

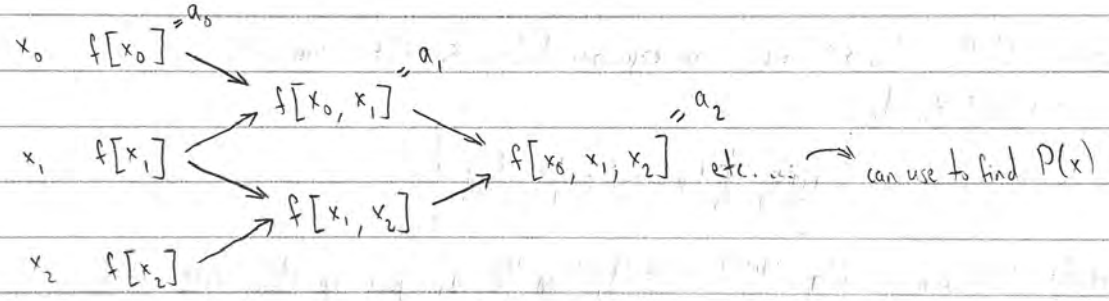
The divided difference polynomial coefficients are given by:

$$a_k = f[x_0, x_1, \dots, x_k]$$

$$\rightarrow \quad P(x) = f(x_0) + \sum_{k=1}^{n} f[x_0, \dots, x_k](x-x_0)(x-x_1)\dots(x-x_{k-1})$$

(*) Algorithm

Can compute higher-order polynomials recursively:



(*) Simplified Form for Equally-Spaced Points

Assume nodes are arranged consecutively with equal spacing $h = x_{i+1} - x_i$ $[i = 0, \dots, n-1]$ between nodes. In this case, can simplify the divided-difference formula: let $x = x_0 + sh$, such that $x - x_i = (s-i)h$ for $i = 0, \dots, n$. Then the polynomial $P(x)$ becomes:

$$P_n[x] = P_n[x_0 + sh] = f[x_0] + shf[x_0, x_1] + s(s-1)h^2 f[x_0, \dots, x_2] + \dots + s(s-1)\dots(s-n+1)h^n f[x_0, \dots, x_n]$$

$$= f[x_0] + \sum_{k=1}^{n} s(s-1)\dots(s-k+1)h^k f[x_0, x_1, \dots, x_k]$$

$$\rightarrow \quad P_n(x) = P_n(x_0 + sh) = f[x_0] + \sum_{k=1}^{n} \binom{s}{k} k! h^k f[x_0, x_1, \dots, x_k]$$

$$\binom{s}{k} = \frac{s(s-1)\dots(s-k+1)}{k!}$$

Runge's Phenomenon

2/5/25

Lecture 13

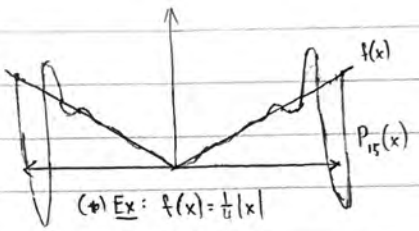
Runge's Phenomenon (Lec. 13)

Previously: saw the Lagrange interpolation error

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)(x-x_1)\dots(x-x_n)$$

Have several issues in polynomial interpolation:

(i) If $f(x) \notin C^{[n+1]}[a,b]$, the n-degree interpolant may be poor [result in a poor approximation].



(*) Ex: $f(x) = \frac{1}{6}|x|$

(ii) Runge's phenomenon: Even if $f \in C^{[n+1]}[a,b]$, the error may still be large if the nodes $\{x_i\}_{i=0,\dots,n}$ are equispaced

Theorem: If the nodes $\{x_i\}_{i=0,\dots,n}$ are equispaced (i.e. $x_i = x_0 + ih$ for $i = 0, \dots, n$) with $x_0 = a$ and $x_n = b$, then:

$$\max_{x\in[a,b]} \prod_{j=0}^{n} |x - x_j| \leq \frac{1}{4} h^{n+1} n!$$

Corollary: Assuming $\max_{x\in[a,b]} |f^{(n+1)}(x)| \leq M$, then the Lagrange interpolation error satisfies:

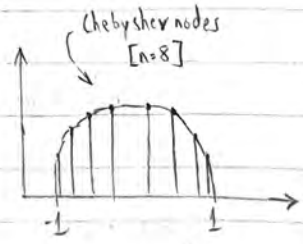
$$|f(x) - P(x)| \leq \frac{M}{(n+1)!} \cdot \frac{1}{4} h^{n+1} n! = \frac{M}{n+1} \cdot \frac{h^{n+1}}{4}$$

(*) Can increase n / decrease h to reduce interpolation error

To reduce error, can choose different sets of nodes (besides equidistant)

→ common choice: Chebyshev nodes ($a = -1$, $b = 1$):

$$\tilde{x}_i = \cos\left(\frac{2i+1}{2n+2}\pi\right), \quad i = 0, \dots, n$$



→ minimizes the error: $\max_{x\in[a,b]} \prod_{j=0}^{n} |x - x_j|$

Hermite Polynomials

2/5/25
Lecture 13 + Lecture 14

Thm: Let $T_n(x) = \prod_{k=0}^{n}(x - \tilde{x}_k)$ be the n^{th}-order Chebyshev polynomial, where $\tilde{x}_k = \cos\left(\frac{2k+1}{2n+2}\pi\right)$, $k = 0, \dots, n$ are the n^{th}-order Chebyshev nodes. Then $T_n(x)$ satisfies:

$$\boxed{T_n(x) = 2^{-n}\cos\left((n+1)\arccos(x)\right)} \quad \forall\, x \in (-1, 1)$$

and $T_n(x)$ satisfies:

$$\boxed{T_{n+1}(x) = xT_n(x) - \frac{1}{4}T_{n-1}(x)}$$

Hermite Polynomials (Lec. 14)

Def: Given $n+1$ distinct points $x_0, x_1, \dots, x_n \in [a, b]$ and non-negative integers $m_0, m_1, \dots, m_n$, let $m = \max\{m_0, m_1, \dots, m_n\}$. Then, given a function $f \in C^m[a, b]$, the osculating polynomial for f at $x_0, x_1, \dots, x_n$ is the polynomial $P(x)$ of least degree satisfying:

$$\boxed{P^{(k)}(x_i) = f^{(k)}(x_i)} \quad \text{for } k = 0, 1, \dots, m_i \text{ for each } i = 0, 1, \dots, n$$

Informally: the osculating polynomial approximating $f \in C^m[a, b]$ is the polynomial of least degree that matches f and its derivatives up to order m_i at each x_i.

- When $n = 0$, the osculating polynomial reduces to the m^{th}-order Taylor polynomial of f at x_0
- The Hermite polynomial is the osculating polynomial corresponding to $m_0 = m_1 = \dots = m_n = 1$
  → matches the value and slope of f at each x_i

Can describe an osculating polynomial by the number of equations / constraints $P^{(k)}(x_i) = f^{(k)}(x_i)$:

$$\#\text{ equations}: \sum_{i=0}^{n}(m_i + 1) \longrightarrow \boxed{\text{degree}[P(x)] \leq (\#\text{ equations}) - 1}$$

→ Hermite polynomials: for $(n+1)$ points, $\deg[P(x)] \leq 2n + 1$.

Cubic Splines

2/10/24
Lecture 14 (cont.)

Cubic Splines (Lec. 14-15)

Previously: saw that Lagrange interpolation can suffer from large oscillations (Runge's phenomenon)
→ in cases where we cannot freely pick sample points $\{x_i\}$, have 2 alternative ways to reduce error:

1. Approximate $f(x)$ using something besides polynomials (ex: Fourier interpolation [151B])
2. Rather than using a global approximation, use piecewise polynomials ← our approach

Def: A cubic spline for a function f defined on $[a,b]$ & with nodes $\{x_j\}_{j=0}^n \subseteq [a,b]$ s.t. $a = x_0 < x_1 < \dots < x_{n-1} < x_n = b$, is a function $S(x)$ satisfying the following:

1. Piecewise Cubic Polynomial: On each sub-interval $[x_j, x_{j+1}]$ for $j = 0, 1, \dots, n-1$, $S(x)$ is a cubic polynomial of the form (for $x \in [x_j, x_{j+1}]$):

$$S(x) = S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

2. Interpolation: $S(x)$ interpolates f at $\{x_j\}$, i.e $S(x_j) = f(x_j)$ for $j = 0, 1, \dots, n$.
3. Continuity & Differentiability: $S(x)$ is continuous and has continuous 1st & 2nd derivatives, i.e $S(x) \in C^2[a,b]$.

Have n subintervals $[x_j, x_{j+1}]$ → $4n$ total coefficients; to get a unique solution, need $4n$ equations as well

From (2), have constraints:

[1] $S_j(x_j) = f(x_j)$ and $S_{j+1}(x_{j+1}) = f(x_{j+1})$ for $j = 0, 1, \dots, n-1$ ← $(n+1)$ equations

From (3), have constraints:

[2] $S_j(x_{j+1}) = S_{j+1}(x_{j+1})$ for $j = 0, \dots, n-2$, and

[3] $S_j'(x_{j+1}) = S_{j+1}'(x_{j+1})$ for $j = 0, \dots, n-2$, and

[4] $S_j''(x_{j+1}) = S_{j+1}''(x_{j+1})$ for $j = 0, \dots, n-2$,

→ $(n-1)$ equations each

→ $(4n-2)$ equations from cubic spline defn.; last 2 equations from boundary conditions

Cubic Splines (cont.)

2/10/25
Lecture 14
+ Lecture 15

Cubic Splines (cont.)

Boundary conditions: constraints on $S(x)$ at the boundary points x_0, x_n

→ have two common choices:

1. **Natural boundary condition**:

$$\boxed{S''(x_0) = 0}, \quad \boxed{S''(x_n) = 0}$$

2. **Clamped boundary condition**: Assuming $f'(x_0)$, $f'(x_n)$ are known:

$$\boxed{S'(x_0) = f'(x_0)}, \quad \boxed{S'(x_n) = f'(x_n)}$$

$4n$ equations for $4n$ unknowns ⇒ obtain a unique cubic spline solution

(*) Proof: Cubic Splines (Existence & Uniqueness)

Idea: Want to explicitly write out the matrix formula for cubic spline coefficients

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

← want to find: a_j, b_j, c_j, d_j $\forall j = 0, \dots, n-1$

From cubic spline conditions:

(i) **Interpolation**: $S_j(x_j) = a_j = f(x_j)$ $\forall j = 0, \dots, n-1$ [gives a_j's]

(ii) **Continuity**: Define $h_j := x_{j+1} - x_j$ $\forall j = 0, \dots, n-1$:

$$S_{j+1}(x_{j+1}) = S_j(x_{j+1}) \Rightarrow a_{j+1} = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3 \quad \forall j = 0, \dots, n-1$$

($a_n = f(x_n)$)

(iii) **1st Derivative**: $b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2$ $\quad \forall j = 0, \dots, n-1$

(iv) **2nd Derivative**: $c_{j+1} = c_j + 3d_j h_j$ $\quad \forall j = 0, \dots, n-1$

(b_n, c_n not used in practice, only for proof)

Via repeated substitution (starting with $d_j h_j = \frac{c_{j+1} - c_j}{3}$), obtain:

$$h_{j-1} c_{j-1} + 2(h_{j-1} + h_j) c_j + h_j c_{j+1} = 3\left(\frac{a_{j+1} - a_j}{h_j} - \frac{a_j - a_{j-1}}{h_{j-1}}\right) \quad \forall j = 0, \dots, n-1$$

a_j, h_j known from $\{x_j\}$, $\{f(x_j)\}$ ⇒ only unknowns are $\{c_j\}_{j=0}^n$; if we can solve for $\{c_j\}$, then we can (via above equations) easily solve for $\{b_j\}_{j=0}^n$, $\{d_j\}_{j=0}^n$.

2/12/25
ecture 15
(cont.)

Cubic Splines (cont.)

(*) Proof (cont.)

Want to show that the following linear system has a solution $\{c_j\}_{j=0}^{n}$

$$h_{j-1}c_{j-1} + 2(h_{j-1}+h_j)c_j + h_j c_{j+1} = 3\left(\frac{a_{j+1}-a_j}{h_j} - \frac{a_j - a_{j-1}}{h_{j-1}}\right) \quad \forall\, j = 0, \dots, n-1$$

Can use the following fact:

> **Lemma:** If a square $n \times n$ matrix M satisfies:
>
> $$|M_{ii}| > \sum_{j \neq i} |M_{ij}|$$
>
> ← M is "strictly diagonally dominant"
>
> then M is invertible.

→ We can write out our formal theorem statement:

> **Theorem:** If f is defined at $a = x_0 < x_1 < \dots < x_n = b$, then f has a unique natural spline interpolant S on the nodes $x_0, x_1, \dots, x_n$ satisfying the natural boundary conditions $S''(a) = 0$, $S''(b) = 0$.

(*) Proof

From boundary conditions, we obtain 2 additional equations $c_0 = 0$, $c_n = 0$.

→ Along with our previous equations, we get a vector equation $Ax = b$ $[A \in \mathbb{R}^{(n+1)\times(n+1)}]$, where:

$$A = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \dots & 0 & 0 \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & & \\ 0 & 0 & \dots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & 0 & \dots & 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 0 \\ \frac{3}{h_1}(a_2-a_1) - \frac{3}{h_0}(a_1-a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n-a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1}-a_{n-2}) \\ 0 \end{bmatrix} \quad x = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix}$$

A is strictly diagonally dominant $\Rightarrow$ by lemma, $\exists$ unique solution for $c_0, c_1, \dots, c_n$. ▨

Note: Can write a similar proof for the clamped boundary condition $[S'(x_0) = f'(x_0),\ S'(x_n) = f'(x_n)]$ as well, just slightly more complicated

Numerical Differentiation

2/14/25
Lecture 16

Numerical Differentiation (Lec. 16-17)

Goal: Want to find methods for numerically approximating derivatives of functions

(*) Motivation: Can use to solve ODEs & PDEs [Math 151B]

Have 2 main approaches for approximating derivatives:

1. Data-based: Given a set of data points $\{(x_i, f(x_i))\}$ and no other knowledge of the function f, can construct an interpolating function (e.g. spline, Lagrange polynomial) and take its derivatives as estimates for f'
2. Taylor expansion: Can use the Taylor expansion to approximate derivatives locally (in some cases: can estimate approximation error) ← us

(i) First-Order Methods

Recall: $f'(x_0) := \lim_{h \to 0} \frac{f(x_0+h) - f(x_0)}{h}$ (Limit defn. of the derivative)

→ Naive approach: Evaluate $\frac{f(x_0+h) - f(x_0)}{h}$ for some h (small) to approximate: (rec: secant method)

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h}$$ (Forward Difference)

$$f'(x_0) \approx \frac{f(x_0) - f(x_0-h)}{h}$$ (Backward Difference)

↖ First-order / "two-point" methods

Can use Taylor's theorem to analyze error:

$$f(x_0+h) = f(x_0) + f'(x_0)h + \frac{f''(\xi)}{2}h^2 \quad \text{(for some } \xi \in [x_0, x_0+h]\text{)}$$

↑ assuming $f \in C^2[a,b]$

$$\Rightarrow \frac{f(x_0+h) - f(x_0)}{h} = f'(x_0) + \frac{h}{2}f''(\xi)$$

Defining $M = \max_{x \in [x_0, x_0+h]} |f''(x)|$:

$$\left| \frac{f(x_0+h) - f(x_0)}{h} - f'(x_0) \right| \leq \frac{h}{2} M$$ (U.B.)

↖ "First-order": error decreases linearly as $h \to 0$

Numerical Differentiation (cont.)

2/14/25
Lecture 16
+Lecture 17

(ii) Second-Order Method

Let $f \in C^3[a,b]$, $x_0 \in [a,b]$; by Taylor's theorem, have that: (for $\xi_1 \in [x_0, x_0+h]$ & $\xi_2 \in [x_0-h, x_0]$)

$$f(x_0+h) = f(x_0) + f'(x_0)h + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f^{(3)}(\xi_1) \quad , \quad f(x_0-h) = f(x_0) - f(x_0)h + \frac{h^2}{2}f''(x_0) - \frac{h^3}{6}f^{(3)}(\xi_2)$$

(nearly) shared terms [besides sign on $f'(x_0)$]

→ Can take difference of two equations to obtain the central difference formula:

$$\frac{f(x_0+h) - f(x_0-h)}{2h} = f'(x_0) + \frac{f^{(3)}(\xi_1) + f^{(3)}(\xi_2)}{12}h^2$$

↩ error is $O(h^2)$ [quadratic]

(iii) Richardson Extrapolation

Goal: Want to use low-order formulas (e.g. forward difference) to generate higher-order results

(*) Notation: Denote the forward difference of f at x_0 (for difference h) by:

$$D_h^+ f(x_0) := \frac{f(x_0+h) - f(x_0)}{h}$$

Looking at $D_{h/2}^+$:

$$D_{h/2}^+ f(x_0) = \frac{f(x_0+h/2) - f(x_0)}{h/2} = \frac{2[f(x_0+h/2) - f(x_0)]}{h}$$

$$\rightarrow 2D_{h/2}^+ f(x_0) = 2f'(x_0) + \frac{h}{2}f''(x_0) + \frac{h^2}{6}f^{(3)}(\xi_1) \quad \text{[via Taylor's thm.]}$$

vs. ($\frac{h}{2}f''(x_0)$: shared term)

$$D_h^+ f(x_0) = f'(x_0) + \frac{h}{2}f''(x_0) + \frac{h^2}{12}f^{(3)}(\xi_2)$$

→ subtracting both equations:

$$\frac{-f(x_0+h) + 4f(x_0+\frac{h}{2}) - 3f(x_0)}{h} = 2D_{h/2}^+ f(x_0) - D_h^+ f(x_0) = f'(x_0) + \frac{1}{6}h^2\left(\frac{1}{2}f^{(3)}(\xi_2) - f^{(3)}(\xi_1)\right)$$

Richardson Extrapolation

2/19/25
Lecture 17 (cont.)

(iii) Richardson Extrapolation (cont.)

Obtained the Richardson extrapolation formula for forward difference:

$$f'(x_0) = \frac{-f(x_0+h) + 4f(x_0+h/2) - 3f(x_0)}{h} + \frac{1}{6}h^2\left(\frac{1}{2}f^{(3)}(x_{\gamma_2}) - f^{(3)}(x_{\gamma_1})\right)$$

($\leftarrow$ error term (3^{rd}-order); $\leftarrow$ from forward diff. [2^{nd}-order])

Notes: (i) Can perform additional iterations to extract higher- and higher-order approximations (via using Taylor system & equations to cancel lower-order terms)

(ii) Can derive similar methods from other differentiation formulas (e.g. central difference) (Taylor-based)

More generally: Richardson extrapolation as a method for accelerating convergence

(*) Round-Off Error

Recall: When working with floating-point, often have to contend with round-off error (esp. when dividing by [small] h, e.g. differentiation)

For differentiation:

$$\tilde{f}(x_0+h) = f(x_0+h) + e(x_0+h), \qquad \tilde{f}(x_0-h) = f(x_0-h) + e(x_0-h)$$

($\tilde{f}(x_0-h)$: FP rep. of $f(x_0-h)$; $e(x_0-h)$: error term)

$$\rightarrow f'(x_0) = \frac{\tilde{f}(x_0+h) - \tilde{f}(x_0-h)}{2h} - \frac{e(x_0+h) + e(x_0-h)}{2h} - \frac{h^2}{6}f^{(3)}(x_\gamma)$$

(computer/FP output; round-off error; Taylor error term)

Assuming round-off errors $e(x_0 \pm h)$ satisfy $|e(x_0 \pm h)| \leq \varepsilon$ [and that $f^{(3)}$ is similarly bounded by $M > 0$], obtain error bound:

$$\left| f'(x_0) - \frac{\tilde{f}(x_0+h) - \tilde{f}(x_0-h)}{2h} \right| \leq \frac{\varepsilon}{h} + \frac{h^2}{6}M$$

($\leftarrow$ as $h \to 0$, ε/h term [round-off error] starts to dominate)

1st term increases as $h \to 0$, whereas 2nd term decreases as $h \downarrow$ $\Rightarrow$ consequently: don't want to reduce h too much

Numerical Integration

2/19/25

Lecture 17 + Lecture 18

Numerical Integration (Lec. 17-21)

Goal: Want a general approach for approximating integrals $\int_a^b f(x)\,dx$, including for functions $f(x)$ with difficult-to-compute (or no) antiderivative

Approach (Numerical Quadrature): We can approximate $\int_a^b f(x)\,dx$ via a discrete weighted sum:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

← weighted sum of $f(x_i)$ [weights w_i]

(*) This is analogous to the Riemann sum:

$$\int_0^1 f(x)\,dx = \lim_{N\to\infty} \sum_{i=1}^{N} f\left(\tfrac{i}{N}\right)\frac{1}{N} \quad \text{[for Riemann-integrable } f\text{]}$$

→ Q: How do we choose the points x_i and weights w_i to minimize approximation error?

The Trapezoidal Rule

(*) Notation: Let $E[f]$ denote the approximation error:

$$E[f] := \int_a^b f(x)\,dx - \sum_{i=0}^{n} w_i f(x_i)$$

Strategy: Approximate $f(x)$ by its Lagrange polynomial $P(x)$

$$f(x) = P(x) + R(x); \quad P(x) = \sum_{i=0}^{n} f(x) L_i(x); \quad R(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\prod_{i=0}^{n}(x - x_i)$$

(R: remainder term;] as seen previously)

$$\to \int_a^b f(x)\,dx = \int_a^b P(x)\,dx + \int_a^b R(x)\,dx = \sum_{i=0}^{n} f(x_i) \underbrace{\int_a^b L_i(x)\,dx}_{w_i} + \underbrace{\int_a^b R(x)\,dx}_{E[f]}$$

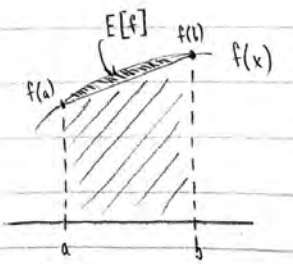
$$\to \boxed{\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} f(x_i)\left[\int_a^b L_i(x)\,dx\right]}$$

In particular: with 2 points x_0 [a], x_0+h [b]:

→ **Trapezoidal Rule:** $\boxed{\int_a^b f(x)\,dx \approx \frac{h}{2} f(a) + \frac{h}{2} f(b)}$

← $E[f] = 0$ for linear f

↑ this is a 2-point method!



The Trapezoidal Rule

2/21/25
Lecture 18
(cont.)

The Trapezoidal Rule (cont.)

Trapezoidal rule error (using Taylor's theorem):

$$E[f] = \int_a^b \frac{f''(x)}{2}(x-a)(x-b) \xrightarrow{[\text{let } f''(x) \le M]} E[f] \le \frac{M}{2}\int_a^b (x-a)(x-b) = \frac{M}{2}\cdot\frac{h^3}{6} \quad [O(h^3)]$$

> **Thm (Weighted MVT):** Let $h \in C(a,b)$ and g integrable on (a,b). If $g(x)$ does not change sign on $[a,b]$, then $\exists\ c \in (a,b)$ s.t.:
>
> $$\int_a^b h(x)g(x)\,dx = h(c)\int_a^b g(x)\,dx$$

In our case: f'', $(x-a)(x-b)$ continuous [and $(x-a)(x-b) \le 0$ for $x \in [a,b]$]: (Trap Rule is 2nd-order!)

$$E[f] = -\frac{f''(c)}{2}\cdot\frac{h^3}{6} \Rightarrow \int_a^b f(x)\,dx = \frac{h}{2}\left[f(a)+f(b)\right] - \frac{h^3}{12}f''(c) \quad [\text{recall: } h = b-a]$$

(*) **Def:** The degree of precision of a quadrature formula is the largest integer n for which the formula is exact for $x^k\ \forall\ k = 0, 1, \dots, n$. Namely: $E[x^k] = 0$ for $k = 0, \dots, n$, but $E[x^{n+1}] \neq 0$.

→ (*) **Ex:** The Trapezoidal Rule is of degree of precision 1

(*) **Remark:** For integrals over long intervals, can divide into subintervals and use Trapezoidal Rule on each
→ total error: (# intervals $\sim 1/h$) · (error per interval: $O(h^3)$) → $O(h^2)$ [2nd-order]

Simpson's Rule

Given 3 points $x_0 = a$, $x_1 = a+h$, and $x_2 = a+2h = b$: assuming $f \in C^4[a,b]$, we can integrate $f(x)$ by approximating f by its Taylor expansion [4th order]:

$$f(x) = \underbrace{f(x_1) + f'(x_1)(x-x_1) + \frac{f''(x_1)}{2}(x-x_1)^2 + \frac{f^{(3)}(x_1)}{6}(x-x_1)^4}_{\text{can approximate [call this } P_3]} + \underbrace{\frac{f^{(4)}(\xi)}{24}(x-x_1)^4}_{\text{error term}}$$

Looking at $\int_a^b P_3(x)\,dx$:

$$\int_a^b P_3(x)\,dx = \int_a^{a+2h}\left[f(x_1) + f'(x_1)(x-x_1) + \frac{f''(x_1)}{2}(x-x_1)^2 + \frac{f^{(3)}(x_1)}{6}(x-x_1)^3\right]dx = 2hf(x_1) + h^3\frac{f^{(3)}(x_1)}{3}$$

Simpson's Rule

2/21/25
Lecture 18
(cont.)

Simpson's Rule (cont.)

Goal: Want to find a "good" approximation P_3 of $f(x)$, s.t.:

$$\int_a^b f(x)\,dx = \int_a^b P_3(x)\,dx + E[f]$$ ← hopefully small

Previously, found a formula in terms of $f''(x_1)$ → now: can use central difference formula to approximate $f''(x_1)$:

$$f''(x_1) = \frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} + \frac{h^2}{12} f^{(4)}(\xi_1) \quad \text{[Central difference formula]}$$

$$\rightarrow \int_{x_0}^{x_2} P_3(x)\,dx = 2hf(x_1) + \frac{h^3}{3}\left(\frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} - \frac{h^2}{12} f^{(4)}(\xi_1)\right) + \frac{h^5}{60} f^{(4)}(c)$$

$$= \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right) + O(h^5)$$

In fact:

$$\boxed{\int_{x_0}^{x_2} f(x)\,dx = \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right) - h^5 \frac{f^{(4)}(\eta)}{90}} \quad \text{[Simpson's Rule]}$$

Notes: (i) Simpson's rule is 4^{th}-order

(ii) Simpson's rule has degree of precision 3 [is exact for $f(x) = 1, x, x^2, x^3$]

(iii) Simpson's rule is a "3-point method"

Newton-Cotes Formulas

2/24/25
Lecture 19

General Newton-Cotes Formulas

Def: Given $(n+1)$ equispaced points $a = x_0 < x_1 < \dots < x_n = b$, the Newton-Cotes formulas approximate the integral of $f(x)$ over $[a, b]$ by:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i) \quad \text{with} \quad w_i = \int_a^b L_i(x)\,dx = \int_a^b \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}\,dx$$

where $L_i(x)$ is the Lagrange interpolating polynomial.

In essence, we simply approximate f by its interpolating polynomial $P(x) = \sum_{i=0}^{n} f(x_i) L_i(x)$:

$$f(x) = P(x) + E(x) \quad \curvearrowright \ (*)\ E(x) = 0 \text{ for } f(x) \in \mathbb{P}_n \ [\deg \leq n]$$

(*) Have 2 "varieties" of Newton-Cotes formulas:

1. Closed N.-C. Formulas: Include evaluations at the endpoints $a = x_0$, $b = x_n$
   - (*) Ex: Trapezoidal rule, Simpson's rule
2. Open N.-C. Formulas: Do not evaluate f at the interval endpoints
   - (*) Ex: Midpoint rule $[n=2 \to \int f(x)\,dx \approx 2hf(x_1)$; 2^{nd}-order$]$

Composite Numerical Integration

Motivation: Although higher-order Newton-Cotes polynomials are theoretically useful, they are very susceptible to Runge's phenomenon in practice [and thus not widely used]

Recall: Previously, looked at 2 different solutions to Runge's phenomenon; can apply the same ideas here:

(i) Pick a "better" set of nodes [not equispaced] ← Gaussian quadrature, later

(ii) Use piecewise polynomials ← composite numerical integration

With Newton-Cotes: at a certain point, error actually increases with # nodes n

→ instead: can divide the integration interval into smaller subintervals, then use a lower-order Newton-Cotes polynomial for each subinterval & sum:

$$a = x_0 < x_1 < \dots < x_n = b \longrightarrow \int_a^b f = \int_{x_0}^{x_1} f + \int_{x_1}^{x_2} f + \dots + \int_{x_{n-1}}^{x_n} f$$

Composite Numerical Integration

2/24/25
Lecture 19
+Lecture 20

Composite Numerical Integration (cont.)

Composite Trapezoidal Rule (Derivation): Applying the Trapezoidal rule to each subinterval $[x_i, x_{i+1}]$:

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,dx = \sum_{i=0}^{n-1} \left[\frac{h}{2} f(x_i) + \frac{h}{2} f(x_{i+1}) - \frac{h^3}{12} f''(c_i) \right]$$

(x_i's equispaced; ← Trapezoidal rule expr.)

By IVT: f'' continuous on $[a, b] \Rightarrow \min_{x\in[a,b]} f'' \le f''(c_i) \le \max_{x\in[a,b]} f''$ for each $i \Rightarrow \exists\, y \in [a, b]$ s.t. $nf''(y) = \sum_{i=0}^{n-1} f''(c_i)$:

→ **Composite Trapezoidal Rule:**

$$\boxed{\int_a^b f(x)\,dx = \frac{h}{2}\left(f(a) + 2\sum_{i=0}^{n-1} f(x_i) + f(b) \right) - \frac{h^2}{12}(b-a) f''(y)}$$

(← using that $h \cdot n = (b-a)$)

Composite Simpson's Rule (Derivation): Given equispaced $a = x_0 < x_1 < \dots < x_n = b$ [for n even]:

$$\int_a^b f(x)\,dx = \sum_{i=1}^{n/2} \int_{x_{2i-2}}^{x_{2i}} f(x)\,dx = \sum_{i=1}^{n/2} \left[\frac{h}{3}\left(f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i}) \right) - \frac{h^5}{90} f^{(4)}(c_i) \right]$$

(c_i in (x_{2i-2}, x_{2i}))

Via IVT (again): $\sum_{i=1}^{n/2} \frac{h^5}{90} f^{(4)}(c_i) \to \frac{h^4}{180}(b-a) f^{(4)}(y)$ for some $y \in [a, b]$:

→ **Composite Simpson's Rule:**

$$\boxed{\int_a^b f(x)\,dx = \frac{h}{3}\left(f(a) + 2\sum_{i=1}^{\frac{n}{2}-1} f(x_{2i}) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + f(b) \right) - \frac{h^4}{180}(b-a) f^{(4)}(y)}$$

(*) notice: $O(h^5)$ [Simpson error], $O(h)$-length subintervals → $O(h^4)$ composite error

Round-Off Error & Stability

Looking at composite Simpson error: assume $f(x_i)$ is approximated by $\tilde{f}(x_i)$ for each i, s.t. $f(x_i) = \tilde{f}(x_i) + e_i$
[e_i: approximation round-off error] → can find total accumulated error $e(h)$ [across all i]:

$$e(h) = \frac{h}{3}\left(|e_0| + 2\sum_{j=1}^{\frac{n}{2}-1} |e_{2j}| + 4\sum_{j=1}^{n/2} |e_{2j-1}| + |e_n| \right)$$

Assuming the round-off errors are uniformly bounded by ε:

$$e(h) \le \frac{h}{3}\left(\varepsilon + 2\frac{n-2}{2}\varepsilon + 4\frac{n}{2}\varepsilon + \varepsilon \right) = \frac{h}{3}(3n\varepsilon) = \underbrace{hn}_{h=\frac{b-a}{n}}\varepsilon = (b-a)\varepsilon$$

→ Unlike (numerical) differentiation: composite quadrature is stable as $h \to 0$

Gaussian Quadrature

2/26/25
Lecture 20

Gaussian Quadrature

Newton-Cotes formulas: use equispaced nodes to approximate integrals

→ Recall: previously, saw that interpolation with Chebyshev nodes resulted in a better Lagrange polynomial (for the same # nodes n); want to do something similar here

Goal (Gaussian Quadrature): Want to find the n nodes $x_1 < x_2 < \dots < x_n$ and corresponding weights $w_1, w_2, \dots, w_n$ to minimize the error in the approximation:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

(since larger degree of exactness generally results in lower error)

In particular: want to maximize the degree of precision/exactness, i.e. ensure the formula is exact for the largest-possible family of polynomials.

(Hope: $2n$ parameters x_i, w_i => resulting formula will be exact for degree $\leq 2n-1$)

Derivation of Gaussian Quadrature

[n=2]: Want to find nodes x_1, x_2 and weights c_1, c_2 such that the formula

$$\int_{-1}^{1} f(x)\,dx \approx c_1 f(x_1) + c_2 f(x_2)$$

is exact for polynomials of degree $2n-1$ [=3]; namely, want to give exact results for $f(x) = 1, x, x^2, x^3$ (on the interval $[-1, 1]$):

$f(x)=1$: $c_1(1) + c_2(1) = \int_{-1}^{1} dx = 2$

$f(x)=x$: $c_1 x_1 + c_2 x_2 = \int_{-1}^{1} x\,dx = 0$

$f(x)=x^2$: $c_1 x_1^2 + c_2 x_2^2 = \int_{-1}^{1} x^2\,dx = 2/3$

$f(x)=x^3$: $c_1 x_1^3 + c_2 x_2^3 = \int_{-1}^{1} x^3\,dx = 0$

Obtain a linear system in terms of c_1, c_2, x_1, x_2 → can solve to find a solution:

$$c_1 = 1,\ c_2 = 1,\ x_1 = -\frac{\sqrt{3}}{3},\ x_2 = \sqrt{3}/3 \longrightarrow \int_{-1}^{1} f(x)\,dx \approx f\left(-\sqrt{3}/3\right) + f\left(\sqrt{3}/3\right)$$

Gaussian Quadrature (cont.)

2/26/25
Lecture 20
+Lecture 21

Derivation of Gaussian Quadrature (cont.)

Can generalize this process to higher-order Gaussian quadrature formulas (i.e more nodes)

Note: Can generalize previous Gaussian quadrature formula (for integrals on $[-1, 1]$) to arbitrary intervals $[a, b]$ via a change of variables:

$$t = \frac{b-a}{2}x + \frac{b+a}{2} \quad \text{for } -1 \le x \le 1$$

such that:

$$\int_a^b f(x)\,dx = \int_{-1}^{1} \overbrace{f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right)}^{g(x)} \frac{b-a}{2}\,dx = \frac{b-a}{2}\int_{-1}^{1} g(x)\,dx$$

↙ integral over $[-1, 1]$

General Gaussian Quadrature

Def. (Orthogonal Polynomials): Define the inner product of functions on $[-1, 1]$ by:

$$\boxed{\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\,dx}$$

→ Two functions f, g are called orthogonal if $\langle f, g\rangle = 0$.

Def. (Legendre Polynomials): The Legendre polynomials are the set of orthogonal polynomials obtained from the Gram-Schmidt process on $\{1, x, x^2, \dots\}$.

(*) Ex: $P_0(x) = 1$; $P_1(x) = x$; $P_2(x) = \frac{3x^2-1}{2}$; $P_3(x) = \frac{5x^3-3x}{2}$

(*) **Recall (Gram-Schmidt):** The vector projection of a vector v onto vector u is defined by:

$$\boxed{\text{proj}_u(v) = \frac{\langle v, u\rangle}{\langle u, u\rangle}\,u}$$

→ Given vectors $v_1, v_2, \dots$, the Gram-Schmidt process finds $u_1, u_2, \dots$ (orthogonal) as:

$$\boxed{u_i = v_i - \text{proj}_{u_1}(v_i) - \dots - \text{proj}_{u_{i-1}}(v_i)} \quad [u_1 = v_1]$$

Gaussian Quadrature (cont.)

2/28/25

Lecture 21 (cont.)

Theorem (Gaussian Quadrature): Let $\{x_i\}_{i=1}^n$ be the roots of the n-degree Legendre polynomial. Assume these roots are real and distinct. Then the Gaussian quadrature rule is:

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where the weights w_i are given by: ($\ell_i(x)$, Lagrange interp. polynomial)

$$w_i = \int_{-1}^{1} \prod_{\substack{j=1 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j} \quad \text{for } i = 1, 2, \ldots, n$$

This formula is exact for any $f \in \mathbb{P}_{2n-1}$, the space of polynomials with degree $\leq 2n-1$.

(*) Proof Sketch

1. Show that $P_n(x)$ [the n^{th} Legendre polynomial] has n roots:
   - Via contradiction: (i) Assume $P_n(x)$ does not have n roots [has $\leq n-1$]
     (ii) Then we can construct $Q(x) \in \mathbb{P}_{n-1}$ s.t. $P_n(x) - Q(x) \geq 0 \Rightarrow \int_{-1}^{1} P_n(x)Q(x)\,dx > 0$
     (iii) But $P_n \perp 1, x, x^2, \ldots, x^{n-1} \Rightarrow P_n \perp Q$, contradiction.
2. Since we are approximating f by an $(n-1)$-degree Lagrange polynomial, we know that our quadrature is exact for $\mathbb{P}_{n-1} = \{\text{polynomials w/ degree} \leq n-1\}$.
3. $\forall f \in \mathbb{P}_{2n-1}$, can write f as $f(x) = p_n(x)q(x) + r(x)$ for some $q, r \in \mathbb{P}_{n-1}$:

$$\rightarrow \int_{-1}^{1} f(x)\,dx = \underbrace{\int_{-1}^{1} p_n(x)q(x)\,dx}_{\text{orthogonal} \Rightarrow = 0} + \int_{-1}^{1} r(x)\,dx = \sum_{i=1}^{n} w_i r(x_i)$$

$$= \sum_{i=1}^{n} w_i \left[p_n(x_i)q(x_i) + r(x_i)\right] = \sum_{i=1}^{n} w_i f(x_i). \quad \blacksquare$$

Remarks (Numerical Integration):

(i) In practice, can split intervals $[a, b]$ into subintervals, use lower-order Gaussian quadrature on each, then combine the results (similar to composite quadrature)

(ii) Quadrature rules extend naturally to higher dimensions (e.g. double integrals)

(iii) The error term for Gaussian quadrature is [for some $\eta \in [a, b]$]:

$$\frac{(b-a)^{2n+1}(n!)^4}{(2n+1)[2n!]^3} f^{(2n)}(\eta)$$

Numerical Integration (cont.)

2/28/25
Lecture 21 (cont.)

(*) Numerical Integration (Wrap-Up)

Computers cannot directly approximate integrals over unbounded domains (i.e. involving either $-\infty$ or $+\infty$ as endpoints)

→ One approach: use a change of variables to transform the domain/variables in order to make the integration bounds finite

(*) Ex: For $\int_0^\infty e^{-x^2}\,dx$, define $z = \frac{x}{1+x}$ (s.t. $z(0)=0$, $z(\infty)=1$):

$$\int_0^\infty e^{-x^2}\,dx \xrightarrow[x\to z]{\text{u-sub}} \int_0^1 e^{-\left(\frac{z}{1-z}\right)^2} \frac{1}{(1-z)^2}\,dz$$

→ integral is over a finite interval ⇒ can approximate numerically

Gaussian Elimination

3/3/25
Lecture 22

Direct Methods for Solving Linear Systems (Lec. 22 - 25)

Goal: Given $A \in \mathbb{R}^{n \times n}$ square matrix and a vector $b \in \mathbb{R}^n$, want to find $x \in \mathbb{R}^n$ satisfying

$$Ax = b, \quad x \in \mathbb{R}^n$$

(*) "$Ax = b$" represents a system of n linear equations w/ n unknowns:

$$\left.\begin{aligned} &E1: a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ &E2: a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ &\vdots \\ &En: a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{aligned}\right\} \iff \sum_{j=1}^{n} a_{ij}x_j = b_i \text{ for } i = 1, 2, \dots, n$$

Additional stipulation: Assume A is invertible ($\det A \neq 0$)
→ a (the) solution x to $Ax = b$ exists & is unique

2 "classes" of methods for solving this problem: ← us, now

1. **Direct methods:** Compute exact solutions in a finite # steps
2. **Iterative methods:** Start with an initial guess & successively refine it to find an approximate solution (see later)
   - Well-suited to solving large linear systems

Gaussian Elimination

Recall (Gaussian Elimination)

Via a 2-phase process:

1. Transform $Ax = b$ into a new system $Ux = y$ (with the same solution x), where U is an upper-triangular matrix:

$$\begin{matrix} E1 \\ E2 \\ \vdots \\ En \end{matrix} \left[\begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{matrix}\right] x = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \longmapsto \left[\begin{matrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & u_{nn} \end{matrix}\right] x = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$ (via row reduction)

2. Solve $Ux = y$ via backward substitution

↳ Can use augmented matrices for notational convenience

3/3/25
ecture 22
Lecture 23

Gaussian Elimination (cont.)

Gaussian Elimination (cont.)

(*) Notation: Can represent the process of eliminating a variable x_i [during (1)] by:

$$\left(E_j - \frac{a_{ji}}{a_{ii}} E_i\right) \to (E_j) \text{ for each } j = i+1, i+2, \dots, n$$

↪ assuming $a_{ii} \neq 0$; otherwise, may need to swap rows

Remark: Gaussian elimination avoids explicitly inverting A [computing A^{-1}] → more efficient/lower computational cost, is the preferred method in many cases
- Inverting A is equivalent to solving $Ax_i = e_i$ for $i = 1, 2, \dots, n$

Note: Even in cases where A is singular or non-square, can still use Gaussian elimination to simplify the overall expression

Computational Cost of Gaussian Elimination

Looking at each step:

1. Variable elimination: To eliminate a variable x_i, require:
   (i) $(n-i)$ divisions to find $m_{ji} := a_{ji}/a_{ii}$ for $j = i+1, i+2, \dots, n$
   (ii) $(n-i)(n-i+1)$ multiplications to compute $m_{ji}E_i$ for $j = i+1, \dots, n$
   (iii) $\underbrace{(n-i)}_{\#\text{ rows}}\underbrace{(n-i+1)}_{\text{entries/row}}$ subtractions to compute $E_j - m_{ji}E_i$ for $j = i+1, \dots, n$

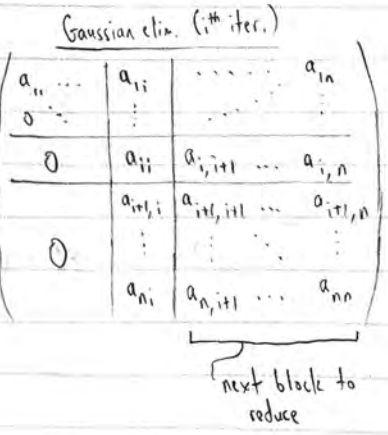
Total # multiplications/divisions:

$$\sum_{i=1}^{n-1} (n-i) + (n-i)(n-i+1) = \frac{2n^3 + 3n^2 - 5n}{6}$$

Total # subtractions:

$$\sum_{i=1}^{n-1} (n-i)(n-i+1) = \frac{n^3 - n}{3}$$

∴ Overall cost of variable elimination: $O(n^3)$



Partial Pivoting

3/5/25
Lecture 23 (cont.)

Computational Cost of Gaussian Elimination (cont.)

Backward substitution: At the i^{th} variable, require:

(i) $(n-i)$ multiplications + 1 division $\longrightarrow$ total: $1 + \sum_{i=1}^{n-1}(n-i+1) = \frac{1}{2}(n^2+n)$

(ii) $(n-i-1)$ additions + 1 subtraction $\longrightarrow$ total: $\sum_{i=1}^{n-1}(n-i-1+1) = \frac{1}{2}(n^2-n)$

In total, across both steps:

- # multiplications / divisions: $\underbrace{\frac{2n^3+3n^2-5n}{6}}_{(1)} + \underbrace{\frac{n^2+n}{2}}_{(2)} = \frac{n^3}{3} - n^2 - \frac{n}{3}$

- # additions / subtractions: $\frac{n^3-n}{3} + \frac{n^2-n}{2} = \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}$

$\rightarrow$ Total cost: $\underbrace{O(n^3)}_{(1)} + \underbrace{O(n^2)}_{(2)} \rightsquigarrow O(n^3)$.

Pivoting Strategies in Gaussian Elimination

Previously: assumed exact arithmetic, could pivot & solve in any order

$\rightarrow$ In practice, round-off errors can affect accuracy of solutions

Gaussian elimination (as given) is susceptible to "bad cases" that may lead to round-off error.

(i) Gaussian elim.: Computes $m_{ji} = a_{ji}/a_{ii}$ $\longrightarrow$ risks dividing by a small number [if a_{ii} small]

(ii) Gaussian elim.: Computes $E_j - m_{ji}E_i$ $\longrightarrow$ risks subtracting 2 nearly-equal numbers (bad)

To remedy (i), can use partial pivoting: instead of choosing the 1st row E_p s.t. $a_{pi} \neq 0$, select the row E_p [$p \geq i$] with largest $|a_{pi}|$, then swap E_p with E_i

$\rightarrow$ Results in less susceptibility to round-off error, in practice (more numerical stability)

(*) Remark: $\exists$ other pivoting strategies, e.g.:

- Scaled partial pivoting: Scale each row by its largest element before pivoting
- Complete pivoting: Search the matrix for its largest element, then swap rows & columns accordingly

LU Decomposition

3/5/25

Lecture 23 + Lecture 24

LU Decomposition

Motivation: Solving $Ax = b$ is very fast $[O(n^2)]$ if A is upper/lower triangular; want to use this property to solve general equations $Ax = b$

> **Strategy: LU Decomposition**
>
> Given an equation $Ax = b$ (for unknown x):
>
> 1. Factor A into $A = LU$, where L, U are lower- and upper-triangular, respectively
> 2. Solve $LUx = b$ for x:
>   (i) Let $y = Ux$, and solve the equation $Ly = b$ for y
>   (ii) Using y, solve $Ux = y$ for x

LU Factorization

Approach: Can express variable elimination $[E_j - m_{ji} E_i \to E_j]$ as a Gaussian transformation matrix:

$$M^{(i)} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & & & & \vdots \\ \vdots & 0 & 1 & & & \vdots \\ \vdots & \vdots & -m_{i+1,i} & 1 & & \vdots \\ \vdots & \vdots & \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & -m_{n,i} & 0 \cdots 0 & 1 \end{pmatrix}$$

↳ eliminates x_i

(notice: this is lower-triangular!)

→ Gaussian elimination can be represented via matrix multiplications:

$$A^{(i)} x = M^{(i-1)} A^{(i-1)} x = M^{(i-1)} b = b^{(i)} \quad (\text{(i)}^{\text{th}} \text{ iteration})$$

→ Final result of Gaussian elimination is an upper-triangular matrix $A^{(n)}$:

$$A^{(n)} x = M^{(n-1)} M^{(n-2)} \cdots M^{(2)} M^{(1)} x = b^{(n)} = M^{(n-1)} M^{(n-2)} \cdots M^{(2)} M^{(1)} b$$

From this, obtain the LU factorization:

$$\underbrace{\left[(M^{(1)})^{-1} (M^{(2)})^{-1} \cdots (M^{(n-1)})^{-1} \right]}_{\text{lower-}\Delta \to L} \overbrace{A^{(n)}}^{\text{upper-}\Delta \to U} x = b \longrightarrow$$

$$L = (M^{(1)})^{-1} (M^{(2)})^{-1} \cdots (M^{(n)})^{-1}$$

$$U = A^{(n)}$$

LU Decomposition (cont.)

3/7/25
Lecture 24 (cont.)

LU Factorization (cont.)

Have the LU factorization: ($L = [M^{(n-1)} \cdots M^{(1)}]^{-1}$)

$$A = LU \quad \text{where} \quad L = (M^{(1)})^{-1} \cdots (M^{(n-1)})^{-1}, \quad U = A^{(n)} = M^{(n-1)} \cdots M^{(1)} A$$

($A = LU$ assuming no row swaps)

Notice: For the transformation matrices $M^{(i)}$, have that:

$$M^{(i)} = \begin{pmatrix} 1 & & & & 0 \\ & \ddots & & & \\ & & 1 & & \\ & & -m_{i+1,i} & \ddots & \\ 0 & & \vdots & & \\ & & -m_{n,i} & 0 & 1 \end{pmatrix} \Longrightarrow (M^{(i)})^{-1} = \begin{pmatrix} 1 & & & & 0 \\ & \ddots & & & \\ & & 1 & & \\ & & +m_{i+1,i} & \ddots & \\ 0 & & \vdots & & \\ & & +m_{n,i} & 0 & 1 \end{pmatrix}$$

→ Obtain a nicer expression for L:

$$\boxed{L = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ m_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & 1 \end{pmatrix}} \qquad A: \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn}^{(n)} \end{pmatrix}$$

Remarks:

(i) The $A = LU$ factorization assumes no row swaps occur; in case of row swaps, instead write $A = P^{-1}LU$, where P is a permutation matrix

(ii) The computational cost of LU factorization is $O(n^3)$, same as Gaussian elimination; once factored, the cost of solving a new linear system $Ax' = b'$ is $O(n^2)$

- LU decomposition is especially useful (over Gaussian elimination) when solving multiple linear systems with the same A but different b

For m equations, using LU decomp. → $O(n^3 + mn)$ instead of $O(mn^3)$

3/7/25
Lecture 24
Lecture 25

Special Matrix Classes

Special Types of Matrices

3 special classes of matrices for which Gaussian elimination or LU factorization can be done efficiently; e.g.:

1. Diagonally-dominant matrices
2. Symmetric positive-definite (SPD) matrices
3. Tridiagonal matrices

(i) Diagonally-Dominant Matrices

Def.: A matrix $A \in \mathbb{R}^{n \times n}$ is called *diagonally dominant* if:

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \quad \text{for all } i = 1, 2, \ldots, n$$

A is *strictly diagonally dominant* if the inequality is strict.

Thm.: Let $A \in \mathbb{R}^{n \times n}$ be strictly diagonally dominant; then:

1. A is nonsingular
2. Gaussian elimination can be performed on any linear system $Ax = b$ without row or column swaps; furthermore, the computations are numerically stable w.r.t. round-off errors

(*) "Computations are numerically stable":

(i) $m_{ji} = a_{ji}/a_{ii}$ has $|m_{ji}| < 1$ [divisor guaranteed to be (relatively) large]

(ii) Since $|a_{ii}| > |a_{ji}|$:

$$a_{ii} - a_{ji}\underbrace{\frac{a_{ij}}{a_{ii}}}_{\text{small}}$$

↘ Two operands will not be close in value
→ no subtraction of nearly-equal numbers
→ less error

The Cholesky Factorization

3/10/25
Lecture 25
(cont.)

(ii) Symmetric Positive-Definite Matrices

Definition: A matrix A is symmetric positive-definite (SPD) if it is symmetric ($A^T = A$) and satisfies:

$$x^T A x > 0 \quad [A \in \mathbb{R}^{n \times n}]$$

for every $x \in \mathbb{R}^n$ s.t. $x \neq 0$.

Theorem: Let $A \in \mathbb{R}^{n \times n}$ be SPD; then:

1. A is invertible
2. Gaussian elimination can be done on $Ax = b$ without row swaps

Theorem (Cholesky Factorization): A matrix A is SPD iff $\exists$ a lower-triangular matrix L with positive diagonal entries s.t. A can be factored as:

$$A = LL^T$$

↳ The Cholesky factorization is equivalent to the LU decomposition for A, i.e. $U = L^T$

Cholesky Decomposition Algorithm

To compute L from A:

1. Set $l_{11} = \sqrt{a_{11}}$
2. For $j = 2, \dots, n$, set $l_{j1} = a_{j1} / l_{11}$
3. For $i = 2, \dots, n-1$:
   (i) Set $l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$
   (ii) For $j = i+1, \dots, n$, set:
   $$l_{ji} = \frac{a_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik}}{l_{ii}}$$
4. Set $l_{nn} = \sqrt{a_{nn} - \sum_{k=1}^{n-1} l_{nk}^2}$

Tridiagonal Matrices

3/10/25
Lecture 25 (cont.)

(iii) Tridiagonal Matrices

Definition: A matrix $A \in \mathbb{R}^{n\times n}$ is called tridiagonal if it is of the form:

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \ddots & \vdots \\ 0 & a_{32} & a_{33} & \ddots & a_{n-1,n} \\ \vdots & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & a_{n,n-1} & a_{nn} \end{pmatrix}$$

Motivation: ODEs can often be expressed (in matrix form) as a tridiagonal matrix
- Can use numerical differentiation methods (e.g. forward difference) to find derivatives f', f'', etc.
  → afterword: left with a (tridiagonal, often) linear system to solve

Remarks:
- Using the sparsity of A, can solve $Ax=b$ [where A is tridiagonal] in only $O(n)$ operations; similarly, solving with forward + backward substitution on $LUx=b$ is also $O(n)$
- The LU factorization of A will have the form:

$$A = LU = \begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \ddots & 0 \\ 0 & l_{32} & l_{33} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & l_{n-1,n} & l_{nn} \end{pmatrix} \begin{pmatrix} 1 & u_{12} & 0 & \cdots & 0 \\ 0 & 1 & u_{23} & \ddots & \vdots \\ 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & u_{n-1,n} \\ 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

The Jacobi Method

3/10/25
Lecture 25

Iterative Methods for Solving Linear Systems (Lec. 25-26)

Iterative methods: To solve a linear system $Ax = b$, start with an initial guess for the solution and progressively refine it to obtain an approximate solution
- Typically define some stopping criteria; in theory, may require an infinite number of iterations to converge to the exact solution

Motivation: Exact methods for solving linear systems (e.g. Gaussian elimination, LU decomp.) can be expensive/require lots of memory for large matrices
- Iterative methods are particularly useful for sparse matrices [most entries 0]

The Jacobi Method

If D, L, U are the diagonal, strictly lower triangular, and strictly upper triangular portions of A, respectively, then the Jacobi method can be written as:

$$\boxed{x^{(k+1)} = D^{-1}\left(b - (L+U)x^{k}\right)}$$

Starting with an initial guess $x^{(0)}$, the Jacobi method computes the next iterate $x^{(k+1)}$ by solving for each component in terms of $x^{(k)}$:

$$\boxed{x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right)} \quad \text{for } i = 1, \ldots, n$$

Properties of the Jacobi Method

- The Jacobi method requires nonzero diagonal entries; if not already true, can permute rows/columns so that the condition is met
- Since the components of the new iterate $x^{(k+1)}$ do not depend on each other, they can be computed in parallel for more performance
- The Jacobi method does not always converge, but has guaranteed convergence under certain conditions (e.g. if A is strictly diagonally dominant); however, the convergence rate may be very slow

3/12/25
Lecture 26

The Gauss - Seidel Method

Gauss - Seidel method: Instead of computing each component of $x^{(k+1)}$ separately, can use earlier components to influence later ones for faster convergence:

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

↳ note: these are computed in sequence → no more parallelism

In terms of D, L, U:

$$x^{(k+1)} = (D+L)^{-1} \left(b - U x^{(k)} \right)$$

Properties of the Gauss - Seidel Method

- Similar to the Jacobi method, Gauss - Seidel also requires nonzero diagonal entries
- Unlike the Jacobi method, Gauss - Seidel can store to $x^{(k+1)}$ in-place ↳ no copying of $x^{(k)}$ needed
- Gauss - Seidel also does not always converge, but has guaranteed convergence under conditions somewhat weaker than those for the Jacobi method (e.g. if A is SPD)
- In practice, Gauss - Seidel converges about 2x as fast as the Jacobi method; however, the rate may still be very slow

The Successive Over - Relaxation (SOR) Method

Successive over-relaxation (SOR) scales the steps taken in Gauss - Seidel by a fixed search parameter ω; namely:

$$x^{(k+1)} = x^{(k)} + \omega \left(x_{GS}^{(k+1)} - x^{(k)} \right)$$

where $x_{GS}^{(k+1)}$ is the next iterate given by Gauss - Seidel, from $x^{(k)}$.
Equivalently:

$$x^{(k+1)} = (1-\omega) x^{(k)} + \omega x_{GS}^{(k+1)}$$

↳ weighted average of $x^{(k)}$, $x_{GS}^{(k+1)}$

Convergence for Iterative Methods

3/12/25
Lecture 26 (cont.)

The SOR Method (cont.)

Can also express the SOR iterate in matrix form:

$$\boxed{x^{(k+1)} = (1-\omega)x^{(k)} + \omega(D+L)^{-1}\left(b - Ux^{(k)}\right)}$$

→ Cases: (i) $\omega = 1$ → Gauss-Seidel (can use to try to make a nonconvergent system converge)
(ii) $\omega < 1$ → Under-relaxation (more cautious/smaller steps)
(iii) $1 < \omega < 2$ → Over-relaxation (more aggressive)
(iv) $\omega > 2$ → Divergence (can use to accelerate convergence, e.g.)

Convergence for Iterative Methods

Definition: Given a matrix $A \in \mathbb{C}^{n\times n}$ with eigenvalues $\lambda_1, \dots, \lambda_n$, its spectral radius is defined by:

$$\boxed{\rho(A) = \max\{|\lambda_1|, |\lambda_2|, \dots, |\lambda_n|\}}$$

Theorem: Let $A \in \mathbb{C}^{n\times n}$ with spectral radius $\rho(A)$; then $\rho(A) < 1$ iff:

$$\boxed{\lim_{k\to\infty} A^k = 0}$$

Otherwise: if $\rho(A) > 1$, $\lim_{k\to\infty} \|A^k\| = \infty$.

With iterative methods on $Ax = b$, have iteration process given by (for some B, g):

$$x^{k+1} = Bx^k + g$$

Let x^* be the exact solution, and define $e^k := x^k - x^*$; we have a recurrence relation:

$$e^{k+1} = Be^k \implies e^k = B^k e^0$$

By the theorem, $e^k \to 0$ for any e^0 iff $B^k \to 0 \iff \rho(B) < 1$; then our criterion is:

→ **Convergence criterion: $\rho(B) < 1$**

The Conjugate Gradient Method

3/12/25
Lecture 26 (cont.)

Conjugate Gradient Method

Optimization methods take the form (when minimizing an objective function ϕ):

$$x^{k+1} = x^k + \alpha^k s_k, \quad \leftarrow \text{"steepest descent" algorithm}$$

where α^k is a parameter chosen to minimize $\phi(x^k + \alpha^k s_k)$ along s_k:

$$\alpha^k = \underset{\alpha}{\operatorname{argmin}}\ \phi(x^k + \alpha s_k)$$

$\uparrow s_k$: step direction (negative gradient, often)

Can consider quadratic functions:

$$\phi(x) = \frac{1}{2} x^T A x - x^T b + c \quad [A \in \mathbb{R}^{n \times n},\ b \in \mathbb{R}^n,\ x \in \mathbb{R}^n]$$

$$\longrightarrow -\nabla \phi(x^k) = b - Ax^k =: r^k$$

Conjugate gradient method: Starting with $s^0 = -\nabla \phi(x^0)$, choose s^k (close to r^k) satisfying:

$$s_k A s_j = 0 \quad \forall\ j = 1, 2, \dots, k-1$$

Then the optimal line search parameter α^k is given by:

$$\boxed{\alpha^k = \frac{r_k^T s_k}{s_k^T A s_k}} \quad [\text{if } s_k \neq 0]$$

Remark: If $A \in \mathbb{R}^{n \times n}$ is SPD, then the quadratic form $\phi(x) = \frac{1}{2} x^T A x - x^T b + c$ attains its minimum at $x^* = A^{-1} b$.

In this case, the conjugate gradient method is guaranteed to converge to x^* within n iterations.