Math 151A Lecture Notes, Fall 2024

Nathan Solomon

December 12, 2024

These notes are meant to be a condensed version of the annotated lecture notes on Canvas for Elisa Negrini's math 151A (applied numerical methods) course. For more detailed notes that cover almost exactly the same content, also check out Kyle Chui's notes: `https://github.com/kylechui/latex/blob/main/05_Winter_2022/Math%20151A/notes.pdf`

Contents

1 Lecture 1

$f \in C([a,b])$ means that f is continuous on $[a,b]$. $f \in C^n([a,b])$ means that f is n times continuously differentiable (that is, $f^{(n)}$ is continuous) on $[a,b]$.

Intermediate Value Theorem (IVT) says that if $f \in C([a,b])$, and there exists $k \in \mathbb{R}$ such that either $f(a) \leq k \leq f(b)$ or $f(b) \leq k \leq f(a)$, then there exists at least one $c \in [a,b]$ such that $f(c) = k$. If f is strictly increasing or strictly decreasing on $[a,b]$, then c is unique.

Taylor's theorem says that if $f \in C^n([a,b])$ and $x_0 \in [a,b]$ and $f^{(n+1)}$ exists (but is not necessarily continuous) on $[a,b]$, then for every $x \in [a,b]$, there exists some $\xi_x \in [x_0, x]$ (or in $[x, x_0]$, which is technically the same) such that $f(x) = P_n(x) + R_n(x)$, where

$$P_n(x) = \sum_{k=0}^{n} \frac{(x-x_0)^k f^{(k)}(x_0)}{k!}$$
$$R_n(x) = \frac{(x-x_0)^{n+1} f^{(n+1)}(\xi_x)}{(n+1)!}.$$

2 Lecture 2

There are 3 types of error that we need to worry about:

- **Computational error** is any error that occurs because computers can only store a finite number of digits. Rounding, truncation, overflow, and underflow are all types of computational error.
- **Truncation error** is error that occurs when you use algorithms that are designed for approximating. For example, approximating a Taylor polynomial with finitely many terms or approximating an integral with a Riemann sum will result in truncation error.
- **Data error** is error that occurs because of noise or measurement error in data. Such errors can be either random or systematic.

The floating point form $Fl(x)$, is the computer's representation of x. We can always write $Fl(x) = x + \varepsilon$, where ε is the rounding error.

When a computer does math, it will either round or chop after each step, so we want to reduce the number of Floating Point Operations (FLOPs). Rewriting an equation in "nested form" can reduce the number of FLOPs, which makes it faster and (usually) more accurate to compute. For example, calculating $x^3 - 6.x^2 + 3.2x + 1.5$ requires 8 FLOPs, but $x(x(x-6.1)+3.2)+1.5$ only requires 5 FLOPs.

If p^* is an approximation of p, then the absolute error is $|p^* - p|$ and the relative error is $|p^* - p| \,/\, |p|$ (assuming $p \neq 0$).

3 Lecture 3

We say that $f(x)$ is $O(g(x))$ as $x \to +\infty$ iff there exist $M, x_0 \in \mathbb{R}$ such that $|f(x)| \leq M\,|g(x)|$ for any $x > x_0$. In other words, $f(x)$ is asymptotically bounded by (a scalar multiple of) $g(x)$.

We say that $f(x)$ is $O(g(x))$ as $x \to a$ iff there exist $M, \delta \in \mathbb{R}$ such that $|f(x)| \leq M\,|g(x)|$ whenever $|x - a| < \delta$.

For practical purposes, you can replace those less-than-or-equal-to signs with equal signs, because it would make sense to call merge sort and FFT "$O(n^2)$" algorithms when you could instead say they are "$O(n \log n)$" algorithms.

A sequence x_n is said to converge with order of convergence $\alpha \geq 1$ to x iff x_n converges to x and there exists $L \in (0, \infty)$ such that

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

L is called the asymptotic error constant.

If $\alpha = 1$ and $L < 1$, then we say x_n converges linearly. If $1 < \alpha < 2$, we say it converges super-linearly, and if $\alpha = 2$, we say it converges quadratically.

4 Lecture 4

Given a function $f \in C([a, b])$ such that $f(a)f(b) < 0$, the bisection method will converge linearly to one root of f. You start by saying $a_1 = a, b_1 = b$, then at each step, define either a_{n+1} or b_{n+1} to be $(a_n + b_n)/2$, and leave the other endpoint unchanged. The error after n steps is less than or equal to $(b-1)/2^n$, and the residual is $|f(p_n)|$, where p_n is our approximation of a root p

5 Lecture 5

A fixed point of a function f is a point p such that $f(p) = p$. Saying that p is a fixed point of f is equivalent to saying p is a root of the function $x \mapsto f(x) - x$.

If $f \in C([a, b])$ and $f(x) \in [a, b]$ for all $x \in [a, b]$, then f has at least one fixed point in $[a, b]$. If those criteria are both true and $f'(x)$ is defined on (a, b) and there exists $k \in (0, 1)$ such that $|f'(x)| \leq k$ for all $x \in (a, b)$, then the fixed point is unique.

Fixed point iteration (FPI) tries to approximate a fixed point p of f by taking an initial guess $p_0 \in [a, b]$ and defining $p_n = f(p_{n-1})$. If the four criteria above are all true, then the sequence p_n will converge to the unique fixed point p for any choice of $p_0 \in [a, b]$, and

$$|p_n - p| \leq k^n \max(p_0 - a, b - p_0)$$

for all n, and

$$|p_n - p| \leq \frac{k^n}{1 - k}\,|p_1 - p_0|$$

for all $n \geq 1$. This implies that p_n converges (at least) linearly to p.

6 Lecture 6

Newton and Rhapson's method (often just called Newton's method) converges to a root quadratically, so it can be much better than the bisection method or fixed point iteration (but it can also fail in some cases).

If $f \in C^2([a, b])$ and we want to find some root p of f, then we choose some initial guess p_0 such that $|p_0 - p|$ is small, then define

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

Newton's method is a local method, meaning it won't converge unless p_0 is sufficiently close to p. One way to get around this is to do a few iterations of a global root-finding method (like the bisection method), then switch to Newton's method.

If we don't have a computationally efficient way to calculate $f'(x)$, then instead of using Newton's method, we can use the secant method, which takes two initial guesses p_0 and p_1, then defines

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

Newton's method can be generalized to work for functions $f : \mathbb{R}^n \to \mathbb{R}^n$, in which case we want to find a vector p such that $f(p) = 0$. For this situation, we define

$$p_{n+1} = p_n - J_f(p_n)^{-1} f(p_n),$$

where J_f is the Jacobian matrix:

$$[J_f(x)]_{i,j} := \frac{\partial f_i(x)}{\partial x_j}.$$

7 Lecture 7

Newton's method converges quadratically ($\alpha = 2$) and the secant method converges super-linearly ($\alpha = \phi = (1 + \sqrt{5})/2 \approx 1.618$, which we will not bother to prove).

First, we want a general theorem for proving how fast FPI converges. If $g \in C^\alpha([a, b])$ for some integer $\alpha \geq 2$ and $p \in [a, b]$ is a point such that $g(p) = p$ and $g'(p) = g''(p) = \cdots = g^{(\alpha-1)}(p) = 0$, but $g^{(\alpha)} \neq 0$, then the sequence defined by $p_{n+1} = g(p_n)$ converges to p with order of convergence α for all p_0 sufficiently close to p. To prove this, write the Taylor series expansion for g centered at p, evaluate it at p_n, and apply Taylor's theorem. You will see that $L = \frac{1}{\alpha!} \lim_{n\to\infty} \left|g^{(\alpha)}(\xi_n)\right|$, where ξ_n is between p and p_n. Therefore $L = \frac{1}{\alpha!} \left|g^{(\alpha)}(p)\right|$.

If we define

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

then

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

which is only defined if $f'(x) \neq 0$. If $f'(x) \neq 0$, then by the above theorem, p_n (defined by $p_{n+1} = g(p_n)$) converges to p with order of convergence $\alpha \geq 2$ for p_0 sufficiently close to p.

8 Lecture 8

A point p such that $f(p) = 0$ is called a zero (of f) of multiplicity m iff there exists a function q such that $f(x) = (x - p)^m q(x)$ for any $x \neq p$, and q is continuous in a neighborhood of m, and $q(p) \neq 0$.

The point $p \in (a, b)$ is a zero of multiplicity m of a function $f \in C^m([a, b])$ iff $0 = f(p) = f'(p) = \cdots = f^{(m-1)}(p)$, but $f^{(m)}(p) \neq 0$.

If $m \geq 1$ and p is a zero of f of multiplicity m, then the function $\mu(x) := f(x)/f'(x)$ has a zero of multiplicity 1 at p. If we want to find a root p of f using Newton's method, but we know that p is a zero (of f) of multiplicity $m > 1$, then we can instead define $p_{n+1} = p_n - \mu(p_n)/\mu'(p_n)$. This should also converge quadratically.

Atkien's Acceleration Theorem says that if p_n converges linearly to p, and $(p_{n+1} - p)(p_n - p) > 0$ for sufficiently large n, then the sequence $\hat{p}_n := p_n - (p_{n+1} - p_n)^2/(p_{n+2} - 2p_{n+1} + p_n)$ satisfies $\lim_{n\to\infty} |\hat{p}_n - p| \,/\, |p_n - p| = 0$, meaning $\hat{p}_n$ converges to p faster than p_n does.

9 Lecture 9

The Weierstrass approximation theorem (also called the Stone-Weierstrass theorem) says that if $f \in C([a, b])$, then for any $\varepsilon > 0$ there exists a polynomial $P(x)$ such that $|f(x) - P(x)| < \varepsilon$ for all $x \in [a, b]$.

Given $n+1$ data points (each data point is a pair $(x_i, f(x_i))$, and $x_i \neq x_j$ unless $i = j$), the Lagrange polynomial is the unique degree n polynomial which goes through all of those points.

Here is an explicit formula for the Lagrange polynomial of degree n:

$$P(x) := \sum_{k=0}^{n} f(x_k) L_{n,k}(x), \qquad L_{n,k}(x) := \prod_{i \in [0,n] \cap \mathbb{Z} - \{k\}} \frac{x - x_i}{x_k - x_i}.$$

Note that $\{L_{n,k}\}$ is basis for the vector space of polynomials of degree n.

10 Lecture 10

If we have $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$ and $x_0 < x_1 < \cdots < x_n$, and $f \in C^{n+1}([a, b])$, and $P(x)$ is the nth degree Lagrange polynomial for f, then for all $x \in [a, b]$, there exists $\xi(x) \in [a, b]$ such that

$$f(x) = P(x) + R(x), R(x) := \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)(x - x_1) \cdots (x - x_n).$$

11 Lecture 11

Neville's method allows you to recursively construct a Lagrange polynomial. If you have a Lagrange polynomial $P_{0,1,\ldots,k}(x)$ which interpolates through points $x_0 < x_1 < \cdots < x_k$, then for any $i \neq j$,

$$P_{0,\ldots,k}(x) = \frac{(x - x_i)P_{0,\ldots,i-1,i+1,\ldots,k}(x) - (x - x_j)P_{0,\ldots,j-1,j+1,\ldots,k}(x)}{x_j - x_i}$$

This polynomial can be constructed by filling in the lower-triangular matrix Q from left to right and top to bottom, or top to bottom and left to right:

$$Q := \begin{bmatrix} P_0(x) & 0 & 0 & \cdots \\ P_1(x) & P_{01}(x) & 0 & \cdots \\ P_2(x) & P_{12}(x) & P_{012}(x) & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Q does not have to be evaluated at x, but if we want to use the Lagrange Polynomial to approximate $f(x)$, then it makes sense to calculate $P_0(x)$, then $P_{01}(x)$, then $P_{012}(x)$, and so on, until adding one more row or column doesn't change our current estimate much – that is, until $|Q_{i,i} = Q_{i+1,i+1}|$ is below our tolerance. It's good to stop before making the degree too high, because we don't want Runge phenomena – that's when the Lagrange polynomial oscillates wildly near the endpoints.

12 Lecture 12

The method of divided differences is another way to recursively calculate the LP. This method ensures it has the form

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0) \cdots (x - x_{n-1}).$$

We define the kth "divided difference" to be $a_k = f[x_0, x_1, \ldots, x_k]$, and define a recursive formula for those constants:

The 0th divided difference, denoted by $f[x_i]$, is defined by $f[x_i] = f(x_i)$. The first divided difference is $f[x_i, x_{i+1}] := (f[x_{i+1}] - f[x_i])/(x_{i+1} - x_i)$. The kth divided difference is

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

If we want to interpolate between $n+1$ points, $\{x_0, x_1, \ldots, x_n\}$ which are equally spaced, with spacing $h := (x_n - x_0)/n$, let $x = x_0 + sh$, so we get

$$\begin{aligned} P(x) &= f[x_0] + \sum_{k=1}^{n} f[x_0, x_1, \ldots, x_k](x - x_0) \cdots (x - x_{k-1}) \\ &= f[x_0] + \sum_{k=1}^{n} f[x_0, x_1, \ldots, x_k] h^k s(s-1) \cdots (x - k + 1) \\ &= f[x_0] + \sum_{k=1}^{n} \binom{s}{k} h^k k! f[x_0, x_1, \ldots, x_k]. \end{aligned}$$

Neville's method is better for evaluating the LP at a point, but the method of divided differences is nicer for finding the coeffiecients of the full LP.

13 Lecture 13

From lecture 10, we have the theorem

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n} (x - x_j).$$

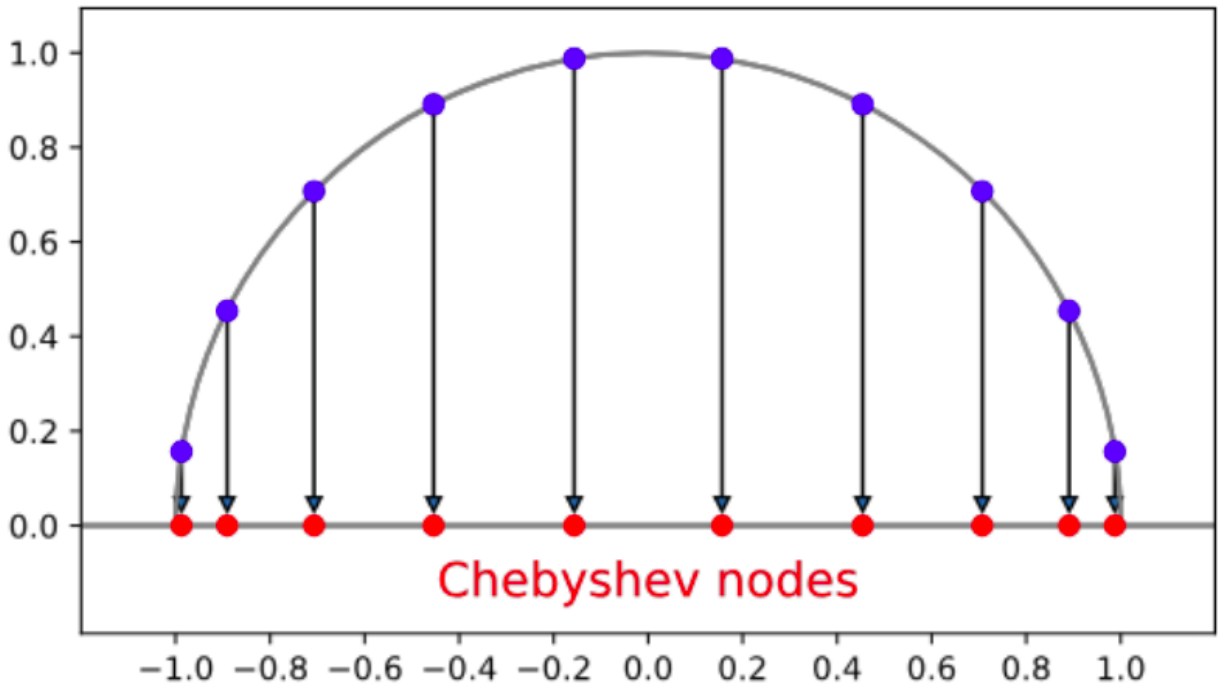
If $M \geq \max_{x \in [a,b]} \left| f^{(n+1)}(x) \right|$ and the interpolation nodes $\{x_0, x_0 + h, x_0 + 2h, \ldots, x_n\}$ are equally spaced, then

$$|f(x) - P(x)| \leq \frac{M}{(n+1)!} \cdot \max_{x \in [a,b]} \prod_{j=0}^{n} |x - x_j| .$$

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

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

To avoid Runge phenomena, we can replace our equally spaced interpolation nodes x_i with Chebyshev nodes, $\widetilde{x_i} := \cos\left(\frac{2i+1}{2n+2}\pi\right), i = 0, \ldots, n$ which get denser closer to the boundary of the domain we are approximating f on.



Of course, if we are interpolating between real data points, we can't choose where the nodes are.

14 Lecture 14

If you have a bunch of nonnegative integers $m_0 < m_1 < \cdots < m_n$, then the osculating polynomial approximating $f \in C^m([a,b])$ is the polynomial P of least degree such that

$$\frac{\mathrm{d}^k P(x_i)}{\mathrm{d}x^k} = \frac{\mathrm{d}^k f(x_i)}{\mathrm{d}x^k}$$

for any $i \in \{0,\ldots,n\}$ and any $k \in \{m_0,\ldots,m_n\}$. The degree of P is at most $M = n + \sum_{i=0}^{n} m_i$. If $\{m_i\} = \{0,1\}$, then we get Hermite polynomials.

Cubic splines are a function S designed to approximate some function f with domain $[a,b]$ by interpolating through nodes $a = x_0 < x_1 < \cdots < x_n = b$.

$$S(x) = \begin{cases} S_0(x) & x_0 \le x \le x_1 \\ S_1(x) & x_1 \le x \le x_2 \\ S_{n-1}(x) & x_{n-1} \le x \le x_n \end{cases}$$

so on the subinterval $[x_j, x_{j+1}]$, $S(x) = S_j(x) = a_j + b_j(x-x_j) + c_j(x-x_j)^2 + d_j(x-x_j)^3$, but we need to solve now for $4n$ constants. We require that $S(x_j) = f(x_j)$ and that $S_j(x_{j+1}) = S_{j+1}(x_{j+1})$. We also require that $S_j'(x_{j+1}) = S_{j+1}'(x_{j+1})$ and $S_j''(x_{j+1}) = S_{j+1}''(x_{j+1})$, but those last two equations are only valid for $j = 0,1,\ldots,n-1$. Therefore we need two more equations, and we have two options for how to get those:

- **Natural/open/free boundary conditions:** $S''(x_0) = 0 = S''(x_n)$.
- **Clamped/closed boundary conditions:** $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$.

15 Lecture 15

Nothing too interesting from this lecture.

16 Lecture 16

For either natural or clamped boundary conditions, spline interpolation exists and is unique. To find it, we can put all the coeffiecients in a matrix and solve. We are guaranteed to have a solution, because a "strictly diagonally dominant" square matrix is invertible.

17 Lecture 17

If we want to approximate the derivative of f at x_0, and the only data we have is $f(x_0)$ and $f(x_0+h)$, then

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

We know that $f''(\xi)$ is bounded as h goes to zero, so the error term is $O(h)$. If $h > 0$, then the "forward difference formula" is

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

and the "backward difference formula" is

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

The error for both of those formulas is $h\left|f''(\xi)\right|/2$ (for some $\xi \in [x_0, x_0+h]$ or $\xi \in [x_0-h, x_0]$, respectively), which is $O(h)$.

We can combine these to get the "center difference formula":

$$f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{h^2}{6}f'''(\xi_x).$$

The error term there depends on some $\xi_x \in [x_0-h, x_0+h]$ which was obtained from $\xi_1 \in [x_0-h, x_0]$ and $\xi_2 \in [x_0, x_0+h]$ using IVT. The error term is $O(h^2)$.

18 Lecture 18

The formulas we are trying to derive for $f^{(n)}(x_0)$ in terms of $f(x)$ at various values of $x \in [x_0-h, x_0+h]$ appear to get better as h goes to zero. But in reality, we will have some rounding/truncation error while computing $f(x)$, which makes the method work worse for very small h. If $\varepsilon \geq \left|f(x)-\widetilde{f}(x)\right|$ is an upper bound on the error in computing $f(x)$ (for x in that interval), then

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

If we have a formula for $f^{(n)}(x_0)$ like the ones described above, we can use Richardson extrapolation to generate another, higher order formula for the same thing. Here's how to do it: take your original formula, and expand it to slightly higher order. Create another version of that formula by replacing every instance of h with $h/2$. Then there is a linear combination of the new formula and the original formula for which the order of the error term is higher than it was before.

19 Lecture 19

One way to numerically integrate a function f is to approximate f with a Taylor polynomial, Lagrange polynomial, or interpolating spline, then integrate the result. Another method is to use Riemann sums. There are several variations of Riemann sums, such as the trapezoidal rule and Simpson's rule. We can generalize those rules with a quadrature formula $Q[f]$. If we want to approximate $\int_{x=a}^{b} f(x)dx$ and we can

only evaluate f at the evaluation nodes $a = x_0 < x_1 < \cdots < x_n = b$, then a quadrature formula $Q[f]$ is defined as

$$Q[f] := \sum_{i=0}^{n} w_i f(x_i),$$

where w_i are weights. We want to choose nodes x_i and weights w_i such that the error,

$$E[f] := \left(\int_a^b f(x)dx \right) - Q[f]$$

is minimized.

If we had computed the Lagrange polynomial for f at the same interpolation nodes x_i, then integrated the Lagrange polynomial exactly, we would get the exact same result as if we had used a quadrature formula with

$$w_i := \int_a^b L_i(x)dx$$

(where L_i is the same as the $L_{i,n}$ defined in lecture 9), and

$$E[f] := \int_a^b \left(\frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \right) dx$$

for some $\xi \in [a, b]$. The formulas here for the weights are called the Newton-Cotes formulas. If there are $n + 1$ interpolation nodes with equal spacing $h = (b - a)/n$, the trapezoidal rule uses 2 nodes with weights $w_0 = h/2, w_1 = h/2$, and has error $-h^3 f^{(2)}(\xi)/12$. Simpson's rule uses 3 nodes with weights $w_0 = h/3, w_1 = 4h/3, w_2 = h/3$ and has error $-h^5 f^{(4)}(\xi)/90$.

The degree of precision of a quadrature formula is defined as the largest positive integer k such that it is exactly correct whenever $f(x)$ is a degree k polynomial function of x. The degree of precision for the trapezoidal rule is one.

20 Lecture 20

Suppose we want to integrate some function f, using only what we know about f at equispaced nodes $x_0, x_1 = x_0 + h, x_2 + 2h$. If we Taylor expand f to order 3, then integrate from x_0 to x_2, we get

$$\int_{x_0}^{x_2} f(x)\mathrm{d}x = 2hf(x_1) + \frac{h^3}{3} f''(x_1) + \frac{1}{24} \int_{x_0}^{x_2} f^{(4)}(\xi(x))(x - x_1)^4 \mathrm{d}x.$$

That last term is the error $E[f]$, and the rest of the expression is $Q[f]$. By the weighted IVT, the error can be rewritten as

$$E[f] = \frac{f^{(4)}(c)h^5}{60}$$

for some $c \in [x_0, x_2]$. If we assume we only know the value of $f(x_0), f(x_1), f(x_2)$, and nothing else, we also have to substitute out $f''(x_1)$ using the centered distance formula:

$$f''(x_1) = \frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} - \frac{h^2}{12} f^{(4)}(\eta)$$

for some $\eta \in [x_0, x_2]$. Using IVT again, there is some $\gamma \in [c, \eta]$ such that

$$\int_{x_0}^{x_2} f(x)\mathrm{d}x = \frac{h}{3} \left(f(x_0) + 4f(x_1) + f(x_2) \right) - \frac{h^5}{90} f^{(4)}(\gamma).$$

Therefore $w_0 = h/3, w_1 = 4h/3, w_2 = h/3$. This method is called Simpson's rule, and it has order $O(h^5)$. It's a 3-point rule with degree of precision 3.

A Newton-Cotes rule which uses the endpoints of each subinterval is called closed, and a Newton-Cotes rule which does not use either of the endpoints is called open. For example, the midpoint rule is open, because integrating f from x_0 to x_2 would require $f(x_1)$, but not $f(x_0)$ or $f(x_2)$.

21 Lecture 21

The trapezoidal rule approximates an integrand with a degree 1 polynomial, and Simpson's rule approximates it with a degree 2 polynomial. Going to higher order would not really be helpful, because then we could get Runge's phenomenon. Instead, we use piecewise polynomials. If we have equally spaced nodes $a = x_0, x_1 = x_0 + h, x_2 = x_0 + 2h, \ldots, x_n = x_0 + nh = b$, then the composite trapezoidal rule uses weights $w_0 = h/2, w_1 = h, w_2 = h, w_3 = h, \ldots, w_{n-1} = h, w_n = h/2$. The error for the composite trapezoidal rule is $-h^2(b-a)f''(\mu)$ for some $\mu \in [a, b]$.

For the composite Simpson's rule, we require n to be even, and use $n/2$ "subintervals. Therefore, the weights are

$$w_0 = \frac{h}{3}, w_1 = \frac{4h}{3}, w_2 = \frac{2h}{3}, w_3 = \frac{4h}{3}, w_4 = \frac{2h}{3}, \ldots, w_{n-1} = \frac{4h}{3}, w_n = \frac{h}{3}$$

and the error is $h^4(b-a)f^{(4)}(\mu)/180$ for some $\mu \in [a, b]$.

In general, using a composite quadrature rule instead of a regular quadrature rule will appear to increase the error from $O(h^n)$ to $O(h^{n-1})$. However, for a composite rule, $h := (b-a)/n$, and for a regular quadrature rule, h is $b - a$ divided by some natural number, so the composite rule is better (which is what we expect, it just looks misleading to write the error in terms of h when the meaning of h has changed).

22 Lecture 22

Gaussian quadrature is the idea that instead of optimizing the weights in order to numerically integrate with equally spaced nodes, we can optimize where to place the nodes and their weights. For this purpose, we will no longer zero-index the nodes and weights, so we will say there are n nodes (labeled $x_1 < x_2 < \cdots < x_n$), and n weights, for a total of $2n$ parameters to be optimized. Therefore it is possible to make the formula exact for any polynomial of degree $2n - 1$. Let GQ_n denote the unique quadrature formula which uses n nodes and has degree of precision $2n - 1$.

The error for GQ_n is

$$E[f] = \frac{(b-a)^{2n+1}(n!)^4}{(2n+1)((2n)!)^2} f^{(2n)}(\xi).$$

Gaussian Quadrature is good at handling indefinite integrals.

GQ_1 is equivalent to the midpoint rule, and GQ_2 has error comparable to Simpson's rule. This table gives the formulas for integrating over $[-1, 1]$ – if you want to integrate over $[a, b]$ instead, the transformation is pretty simple.

Formula	x_i	w_i	Error bound
GQ_1	0	2	$\frac{2}{3}f''(\xi)$
GQ_2	$-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}$	$1, 1$	$\frac{8}{45}f^{(4)}(\xi)$
GQ_3	$-\sqrt{\frac{3}{5}}, 0, \sqrt{\frac{3}{5}}$	$\frac{5}{9}, \frac{8}{9}, \frac{5}{9}$	$\frac{8}{175}f^{(6)}(\xi)$

23 Lecture 23

When you want to solve a matrix equation $Ax = b$, "Gaussian elimination with backward substitution" is the method where you use elementary row operations to rewrite the equation as $Ux = y$, where x is unchanged, and U is upper-triangular. Then you can look at the system of equation described by that, and solve for elements of x by going through those equations from bottom to top.

Here is the algorithm we are expected to follow for Gaussian elimination:

- Start with the augmented matrix $[A|b]$
- For $i = 1, 2, \ldots, n-1$, let p be the smallest integer $i \leq p \leq n$ such that $a_{p,i} \neq 0$
- If $p \neq i$, swap rows i and p
- For each j in $i+1, i+2, \ldots, n+1$, define $\lambda_{i,j} = -a_{j,i}/a_{i,i}$ and add $\lambda_{i,j}$ times row i to row j.

- Now you should be left with an upper triangular matrix, so you can use backwards substitution to solve for x

24 Lecture 24

If A is singular, $Ax = b$ has either zero or infinitely many solutions, and if it has infinitely many solutions, Gaussian elimination with backwards subsitution will find them one of them, although you will have to arbitrarily choose a free variable.

That method runs in $O(n^3)$ time, and it can be very vulnerable to round-off error. The principle of partial pivoting tries to mitigate that, by making the "pivots" (that is, the number $a_{i,i}$ in the algorithm description from the last lecture) as large as possible. Instead of letting p be the smallest integer between i and n such that $a_{p,i}$ is nonzero, let p be the integer between i and n which maximizes $a_{p,i}$.

25 Lecture 25

LU decomposition is a trick that helps you solve $Ax = b$ for any b, for a given A. The goal is to find a lower-triangular matrix L and an upper-triangular matrix U such that $A = LU$. Once you have done that, you can let $y = Ux$, then use forward substitution to solve $Ly = b$ for y, then use backward substitution so solve $Ux = y$ for x. Forward and backward substitution take $O(n^2)$ time, so the hard part (that is, the LU decomposition, which takes $O(n^3)$ time) only needs to be done once.

To find the LU decomposition, use Gaussian elimination to turn A into U, but keep track of the elementary row operation needed to do that. The product of those row operations is L^{-1}. In other words, if you have an augmented matrix $[A|I]$, then row operations will turn it into $[U|L^{-1}]$.

By convention, L has ones on the diagonal.

26 Lecture 26

An $n \times n$ matrix A is called "diagonally dominant" iff

$$|A_{ii}| \geq \sum_{j=1, j \neq i}^{n} |A_{ij}|$$

for every i, and "strictly diagonally dominant" if that is still true when you replace $\geq$ with $>$.

Any strictly diagonally dominant matrix A is non-singular, because $Ax = b$ can always be solved without any row or column swaps.

A is called "symmetric positive semi-definite" iff it is symmetric ($A = A^T$) and for amy $x \in \mathbb{R}^n, x \neq 0$, $x^T Ax \geq 0$, and "symmetric positive definite" if that is still true when you replace $\geq$ with $>$.

The Cholesky factorization theorem says that A is symmetric positive definite iff there exists a lower triangular matrix L such that $A = LL^T$ (which is called the Cholesky factorization). Here is the algorithms to find L:

- Let $L_{11} = \sqrt{A_{11}}$
- For j from 2 to n, set $L_{j1} = A_{j1}/L_{11}$
- For i from 2 to $n-1$, set

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

  and for j from $i+1$ to n, set

$$L_{ji} = \left(A_{ji} - \sum_{k=1}^{i-1} L_{jk} L_{ik} \right) / L_{ii}$$

- Set $L_n n = \sqrt{A_{nn} - \sum_{k=1}^{n-1} L_{nk}^2}$

If we know A is tridiagonal, we can solve $Ax = b$ in $O(n)$ time.