Numerical Analysis Notes

A Study on Computational Methods and Error Analysis

Yifan Wang
Department of Mathematics and Statistics
Texas Tech University

November 5, 2025

Contents

1 Computer Arithmetic and Error Analysis

1.1 Review of Calculus

1.1.1 Intermediate Value Theorem (IVT)

If f is continuous on the closed interval $[a, b]$ and N is any number between $f(a)$ and $f(b)$, where $f(a) \neq f(b)$, then there exists at least one $c \in (a, b)$ such that

$$f(c) = N.$$

In particular, if $f(a)f(b) < 0$, then there exists $c \in (a, b)$ such that $f(c) = 0$. This forms the basis for the *bisection method* for finding roots:

$$f(a)f(b) < 0 \quad \Rightarrow \quad \exists\, c \in (a, b), \text{ such that } f(c) = 0.$$

1.1.2 Taylor's Theorem

If f has $n + 1$ continuous derivatives near x_0, then for any x near x_0:

$$f(x) = P_n(x) + R_n(x),$$

where

$$P_n(x) = f(x_0)+f'(x_0)(x-x_0)+\frac{f''(x_0)}{2!}(x-x_0)^2+\cdots+\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n = \sum_0^n \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n,$$

and the Lagrange form of the remainder is:

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}, \quad \text{for some } \xi \text{ between } x_0 \text{ and } x.$$

Special Case: Mean Value Theorem (MVT)

For $n = 0$ (e.g., $f(x) = P_0(x) + R_0(x)$):

$$f(x) = f(x_0) + f'(\xi)(x - x_0),$$

which gives the classical Mean Value Theorem. In particular, for $a < b$:

$$f(b) = f(a) + f'(\xi)(b - a), \quad \text{where } \xi \in (a, b).$$

Thus,

$$f'(\xi) = \frac{f(b) - f(a)}{b - a}.$$

Example 1: Find $P_2(x)$ for $f(x) = e^x \cos x$ at $x_0 = 0$

We need the second-degree Taylor polynomial around $x_0 = 0$:

$$P_2(x) = f(0) + f'(0)\, x + \frac{f''(0)}{2}\, x^2.$$

Step 1: Compute derivatives at $x = 0$

$$f(x) = e^x \cos x, \quad f(0) = e^0 \cos 0 = 1.$$

$$f'(x) = e^x(\cos x - \sin x), \quad f'(0) = 1(1-0) = 1.$$

$$f''(x) = \frac{d}{dx}\big(e^x(\cos x - \sin x)\big) = e^x(\cos x - \sin x) + e^x(-\sin x - \cos x) = -2e^x \sin x.$$

Hence,

$$f''(0) = -2e^0 \sin 0 = 0.$$

For later use in the error term $R_2(x)$, we compute $f^{(3)}(x)$:

$$f^{(3)}(x) = \frac{d}{dx}(-2e^x \sin x) = -2e^x \sin x - 2e^x \cos x = -2e^x(\sin x + \cos x).$$

Step 2: Substitute into Taylor polynomial

$$P_2(x) = f(0) + f'(0)\, x + \frac{f''(0)}{2}x^2 = 1 + x + 0 = 1 + x.$$

Error Term $R_2(x)$ can be explicitly expressed by Taylor's theorem:

$$R_2(x) = \frac{f^{(3)}(\xi)}{3!}x^3, \quad \text{for some } \xi \in (0, x).$$

Using the expression for $f^{(3)}(x)$:

$$R_2(x) = \frac{-2e^{\xi}(\sin \xi + \cos \xi)}{3!}\, x^3 = -\frac{e^{\xi}(\sin \xi + \cos \xi)}{3}\, x^3, \quad \xi \in (0, x).$$

Final Answer

$$\boxed{P_2(x) = 1 + x, \qquad R_2(x) = -\frac{e^{\xi}(\sin \xi + \cos \xi)}{3}\, x^3, \quad \xi \in (0, x).}$$

Note: The quadratic term vanishes since $f''(0) = 0$. The remainder $R_2(x)$ is of order x^3, and becomes small near $x = 0$.

Example 2: Find the error bound at x=0.01 for $f(x) = e^x \cos x$, $x_0 = 0$

From the Taylor expansion about $x_0 = 0$ with remainder,

$$f(x) = 1 + x - \frac{e^{\xi}(\sin \xi + \cos \xi)}{3}\, x^3, \qquad \xi \in (0, x).$$

At $x = 0.01$,

$$e^{0.01} \cos 0.01 = 1 + 0.01 - \frac{e^{\xi}(\sin \xi + \cos \xi)}{3}\, (0.01)^3 = 1.01 - \frac{e^{\xi}(\sin \xi + \cos \xi)}{3} \times 10^{-6}, \quad \xi \in (0, 0.01).$$

Approximation and remainder

The second-degree Taylor polynomial gives the approximation

$$P_2(0.01) = 1.01,$$

and the truncation error is

$$R_2(0.01) = -\frac{e^{\xi}(\sin\xi + \cos\xi)}{3} \times 10^{-6}, \qquad \xi \in (0, 0.01).$$

Crude bound

Using $|\sin\xi + \cos\xi| \le \sqrt{2}$ and $e^{\xi} \le e^{0.01}$,

$$|R_2(0.01)| \le \frac{e^{0.01}\sqrt{2}}{3} \times 10^{-6} \approx 4.76 \times 10^{-7}.$$

Sharper bound for small ξ

Since $0 \le \xi < 0.01$, we have $|\sin\xi| \le \xi \le 0.01$ and $|\cos\xi| \le 1$. Hence

$$|\sin\xi + \cos\xi| \le |\sin\xi| + |\cos\xi| \le 0.01 + 1 = 1.01,$$

so

$$|R_2(0.01)| \le \frac{e^{0.01} \cdot 1.01}{3} \times 10^{-6} \approx 3.40 \times 10^{-7}.$$

Numerical enclosure

Therefore

$$|e^{0.01}\cos 0.01 - 1.01| \;\le\; 3.40 \times 10^{-7},$$

and

$$1.01 - 3.40 \times 10^{-7} \;\le\; e^{0.01}\cos 0.01 \;\le\; 1.01 + 3.40 \times 10^{-7},$$

i.e.

$$\boxed{1.00999966 \;\lesssim\; e^{0.01}\cos 0.01 \;\lesssim\; 1.01000034.}$$

Remark. The sharper bound mirrors the cosine example's improvement (replacing the crude bound on the remainder by information specific to small ξ), and it is close to the actual error $|\, e^{0.01}\cos(0.01) - 1.01\,| \approx 3.35 \times 10^{-7}$.

Any other approaches?

Given the exponential function e^x has a Maclaurin series expansion (Taylor series around $x_0 = 0$) that converges for all real and complex x, and the Maclaurin series expansion for $\cos x$:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots, \qquad \cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots.$$

Multiplying and collecting terms (Maclaurin for $e^x \cos x$):

$$e^x \cos x = 1 + x + 0 \cdot x^2 \; - \frac{x^3}{3} - \frac{x^4}{6} - \frac{x^5}{30} + O(x^6).$$

Lest take the Fourth-degree approximation (much tighter), using the series up to x^4,

$$P_4(x) = 1 + x - \frac{x^3}{3} - \frac{x^4}{6},$$

so

$$P_4(0.01) = 1 + 0.01 - \frac{(0.01)^3}{3} - \frac{(0.01)^4}{6} = 1.009999665.$$

The next term in the series is $-\dfrac{x^5}{30}$, so the truncation error is

$$R_4(x) = \frac{f^{(5)}(\xi)}{5!}x^5, \qquad \text{with } \frac{f^{(5)}(0)}{5!} = -\frac{1}{30}, \;\; \xi \in (0, x).$$

At $x = 0.01$, the magnitude of the next-term estimate is

$$\left|\frac{x^5}{30}\right| = \frac{10^{-10}}{30} \approx 3.33 \times 10^{-12},$$

hence

$$\boxed{e^{0.01}\cos(0.01) \approx 1.009999665 \quad \text{with error } \lesssim 3.33 \times 10^{-12}.}$$

Remarks. (i) P_2 already gives 6 correct digits for this x; P_4 is far more accurate (error $\sim 10^{-12}$). (ii) The absence of the x^2 term (since $f''(0) = 0$) explains the unusual small error.

1.1.3 Bisection method and Secant method

Bisection Method

It is a simple and robust numerical technique for determining a root of a continuous function $f(x)$ within a closed interval $[a, b]$. The method assumes that $f(a)$ and $f(b)$ have opposite signs, that is

$$f(a)\, f(b) < 0,$$

so that by the Intermediate Value Theorem there exists at least one root $\xi \in [a, b]$ with $f(\xi) = 0$.

The procedure begins by selecting an initial interval $[a, b]$ satisfying the sign change condition. The midpoint of the interval is then computed as

$$c = \frac{a + b}{2}.$$

Next, the function is evaluated at c. If $f(c) = 0$, then c is exactly the root. If $f(c)\, f(a) < 0$, the root lies in the subinterval $[a, c]$ and we replace b by c. Otherwise, the root lies in $[c, b]$ and we replace a by c. This process is repeated until the length of the interval $|b - a|$ becomes smaller than a prescribed tolerance or a maximum number of iterations is reached.

Secant Method

In contrast to the bisection method, it does not require that the function changes sign over an interval. Instead, the derivative is approximated by the slope of the secant line through two successive points on the graph of f.

Starting with two initial guesses x_0 and x_1, the method constructs the secant line joining the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$. The point where this line intersects with the x-axis is taken as the next approximation of the root. This leads to the iterative formula as follows:

$$x_{k+1} = x_k - f(x_k)\, \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}.$$

The process generates a sequence $x_0, x_1, x_2, \ldots$ which, under appropriate conditions, such as continuity of f and sufficiently good initial approximations, converges to the actual root.

Algorithm 1 Bisection Method for Root Finding

Require: Function $f(x)$, initial interval $[a, b]$ with $f(a) \cdot f(b) < 0$, tolerance ε, maximum iterations $N_{\max}$

Compute $f(a)$ and $f(b)$
for $k = 1$ to $N_{\max}$ **do**
  Compute midpoint:

$$c = \frac{a + b}{2}$$

  Evaluate $f(c)$
  if $|f(c)| < \varepsilon$ **or** $\frac{|b-a|}{2} < \varepsilon$ **then**
    Return c as the root and stop
  end if
  if $f(a) \cdot f(c) < 0$ **then**
    $b \leftarrow c$
  else
    $a \leftarrow c$
  end if
end for
Output: Method did not converge within $N_{\max}$ iterations

Algorithm 2 Secant Method for Root Finding

Require: Function $f(x)$, initial guesses x_0, x_1, tolerance ε, maximum iterations $N_{\max}$

Compute $f_0 = f(x_0)$ and $f_1 = f(x_1)$
for $k = 1$ to $N_{\max}$ **do**
  Compute next approximation using the secant formula:

$$x_{k+1} = x_k - f(x_k) \cdot \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

  Evaluate $f_{k+1} = f(x_{k+1})$
  if $|x_{k+1} - x_k| < \varepsilon$ **or** $|f_{k+1}| < \varepsilon$ **then**
    Return x_{k+1} as the root and stop
  end if
end for
Output: Method did not converge within $N_{\max}$ iterations

The secant method generally converges faster than the bisection method and exhibits superlinear convergence, although it can fail if the denominator $f(x_k) - f(x_{k-1})$ becomes very small or if the initial guesses are far from the root.

Ramberge iteration (find all zeros of a function)

Ramberge iteration is designed to locate all zeros of a polynomial simultaneously, rather than one at a time. More details can be found in Burden's numerical analysis book.

1.1.4 Rolle's Theorem (a special case of the Mean Value Theorem)

Suppose $f \in C[a, b]$ and f is differentiable on (a, b). If $f(a) = f(b)$, then a number c in (a, b) exists with $f'(c) = 0$.

Intuition: If the function starts and ends at the same value and is smooth (continuous and differentiable), then its graph must have at least one horizontal tangent somewhere between a and b.

Generalized Rolle's Theorem

Suppose $f \in C[a, b]$ (i.e., f is continuous on $[a, b]$) and f is n times differentiable on (a, b). If

$$f(x) = 0, \quad \text{at } n+1 \text{ distinct points } a \le x_0 < x_1 < \cdots < x_n \le b,$$

then there exists a number $c \in (x_0, x_n) \subset (a, b)$ such that

$$f^{(n)}(c) = 0.$$

Intuition: This is a generalization of the classical Rolle's Theorem (which corresponds to $n = 1$). If a function has $n+1$ distinct zeros and is sufficiently smooth, its n-th derivative must vanish somewhere between the first and last zero.

Idea of Proof: The proof is by induction:

1> For $n = 1$: This is exactly Rolle's Theorem. If $f(x_0) = f(x_1) = 0$ with $x_0 < x_1$, then there exists $c \in (x_0, x_1)$ such that $f'(c) = 0$.

2> Induction Step: Assume the statement is true for $n-1$. Suppose f has $n+1$ distinct zeros $x_0 < x_1 < \cdots < x_n$. Apply the classical Rolle's Theorem to f on each subinterval $[x_{k-1}, x_k]$ for $k = 1, \ldots, n$. For each such interval, there exists a point $y_k \in (x_{k-1}, x_k)$ where $f'(y_k) = 0$. Thus, f' has n distinct zeros $y_1 < y_2 < \cdots < y_n$ in (x_0, x_n). By the induction (hypothesis applied to f', which is $n-1$ times differentiable, the statment with the assumption is true for $n-1$), there exists $c \in (y_1, y_n) \subset (x_0, x_n)$ such that

$$f^{(n)}(c) = 0.$$

Example 3: Consider $f(x) = x(x-1)(x-2)$ on $[0, 2]$. Here $n = 2$ and zeros are at $x_0 = 0, x_1 = 1, x_2 = 2$. We have $f''(x) = 6x - 6$, and $f''(1) = 0$. Thus $f^{(2)}(c) = 0$ at $c = 1$, confirming the theorem.

1.2 Floating-point systems; rounding error; propagation of error

1.2.1 IEEE 754 binary64

A double-precision (64-bit) floating-point number has

$$(-1)^s\, 2^{\,c-1023}\,(1+f),$$

where s is the sign bit, $c \in \{1,\dots,2046\}$ is the 11-bit exponent (biased by +1023), and f is the 52-bit fraction

$$f = \sum_{k=1}^{52} b_k\, 2^{-k} \quad \text{with } b_k \in \{0,1\}.$$

(Exponent $c = 0$ and $c = 2047$ are reserved for subnormals/zero and $\pm\infty$, NaN, respectively.)

Example 1: Given bit string

$$\underbrace{0}_{s}\ \underbrace{10000000011}_{\text{exponent } c}\ \underbrace{1011100100010000000000000000000000000000000000000000}_{\text{fraction bits}}$$

Step 1: Sign and exponent

Sign bit: $s = 0 \Rightarrow (-1)^s = +1$.

Exponent bits 10000000011 give

$$c = 1\cdot 2^{10} + 0\cdot 2^9 + \cdots + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 1024 + 2 + 1 = 1027.$$

Unbiased exponent: $c - 1023 = 1027 - 1023 = 4$.

Step 2: Fraction (mantissa) f

The first 16 fraction bits are $b_1 b_2 \dots b_{16} = 1011100100010000$, followed by zeros. Hence the nonzero positions are $k \in \{1,3,4,5,8,12\}$, so

$$f = 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12} = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{4096} = \frac{2961}{4096}.$$

Therefore the significand is $1 + f = \dfrac{7057}{4096}$.

Step 3: Value

$$x = (-1)^s\, 2^{\,c-1023}\,(1+f) = 1\cdot 2^4 \cdot \frac{7057}{4096} = 16\cdot\frac{7057}{4096} = \frac{7057}{256} = 27.56640625.$$

Therefore, the 64-bit pattern

$$0100000000111011100100010000000000000000000000000000000000000000$$

represents the real number $\boxed{27.56640625}$ in IEEE 754 binary64 (normalized) form.

Let's do it reversely, give you the number 27.56640625, convert it to binaries.

Step 1: Number is positive s=0.

Step 2: Deal the integer part 27 (divide by 2 repeatedly and record the remainders from bottom to top): $27_{10} = 11011$

Step 3: Deal the fractional part (multiply by 2 repeatedly and record the interger part from top to bottom)

$$
\begin{aligned}
&0.56640625 \times 2 = 1.1328125 \Rightarrow \text{ digit } = 1, \text{ remainder } = 0.1328125\\
&0.1328125 \times 2 = 0.265625 \Rightarrow \text{ digit } = 0, \text{ remainder } = 0.265625\\
&0.265625 \times 2 = 0.53125 \Rightarrow \text{ digit } = 0, \text{ remainder } = 0.53125\\
&0.53125 \times 2 = 1.0625 \Rightarrow \text{ digit } = 1, \text{ remainder } = 0.0625\\
&0.0625 \times 2 = 0.125 \Rightarrow \text{ digit } = 0, \text{ remainder } = 0.125\\
&0.125 \times 2 = 0.25 \Rightarrow \text{ digit } = 0, \text{ remainder } = 0.25\\
&0.25 \times 2 = 0.5 \Rightarrow \text{ digit } = 0, \text{ remainder } = 0.5\\
&0.5 \times 2 = 1.0 \Rightarrow \text{ digit } = 1, \text{ remainder } = 0.0 \rightarrow \text{ stop .}
\end{aligned}
$$

therefore

$$0.56640625_{10} = 0.10010001$$

Step 4: Shift the binary point so it's 1 .(fraction) $\times 2^e$:

$$11011.10010001_2 = 1.101110010001_2 \times 2^4$$

which gives the exponent $e = 4$. Namely $c = e + 1023 = 1027 = 10000000011$

Step 5: Combine.

1.2.2 Neighboring machine numbers

Given the 64-bit pattern from before (27.56640625), the next smaller and next larger representable (normalized) numbers are the bit patterns

$$\text{nextdown: } 0100000000111011100100000\underbrace{111\cdots111}_{\text{52 fraction bits all 1}},$$

$$\text{nextup: } 0100000000111011100100001\underbrace{000\cdots00}_{\text{51 zeros}}1$$

.

These two are the immediate neighbors of 27.56640625 in the set of IEEE 754 binary64 numbers.

This means that our original machine number represents not only 27.56640625 , but also half of the real numbers that are between 27.56640625 and the next smallest machine number, as well as half the numbers between 27.56640625 and the next largest machine number. To be precise, it represents any real number in the interval

$$
\begin{aligned}
&[27.566406249999998223643160599749535322189331054687 5,\\
&27.566406250000001776356839400250464677810668945312 5).
\end{aligned}
$$

Definition: ulp(x) (unit in the last place) is the spacing between consecutive representable floating–point numbers near x.

Smallest and largest positive numbers

Ignoring subnormals (less precision than a normal float point):

$$\text{min normalized: } s=0,\ c=1,\ f=0 \ \Rightarrow\ 2^{-1022}(1+0)\ \approx\ 2.2251\times 10^{-308},$$

$$\text{max finite: } s=0,\ c=2046,\ f=1-2^{-52} \ \Rightarrow\ 2^{1023}\big(2-2^{-52}\big)\ (f's\ all\ bits=1)\approx\ 1.7977\times 10^{308}.$$

Underflow and overflow

If, in a computation, $|x|$ exceeds the largest finite value $2^{1023}(2-2^{-52})$, an *overflow* occurs (typically raising an exception). If $|x|$ is smaller than the smallest *normalized* value 2^{-1022}, you enter the *subnormal* range (see note below); values smaller than the smallest positive subnormal are flushed to zero (*underflow to zero*).

Signed zeros

There are two encodings for zero:

$$+0:\ s=0,\ c=0,\ f=0, \qquad -0:\ s=1,\ c=0,\ f=0.$$

They compare equal as real numbers but can matter in directed rounding and certain numerical algorithms.

1.2.3 Absolute vs. Relative Error

Given a true value p and an approximation p^*,

$$\text{absolute error} = \big|p-p^*\big|, \qquad \text{relative error} = \frac{\big|p-p^*\big|}{|p|} \quad (p\neq 0).$$

Example 2: $f(x)=e^x\cos x$ at $x=0.01$

Let

$$p=f(0.01)=e^{0.01}\cos(0.01)\approx 1.0099996649966665.$$

Approximation 1 (degree-2 Taylor): $p^*=P_2(0.01)=1+0.01=1.01.$

$$\big|p-p^*\big| = \big|\,1.0099996649966665-1.01\,\big| \approx 3.350033335181024\times 10^{-7},$$

$$\frac{\big|p-p^*\big|}{|p|}\approx \frac{3.350033335181024\times 10^{-7}}{1.0099996649966665}\approx 3.316865788457545\times 10^{-7}.$$

Interpretation: relative error $\sim 10^{-7}$ (about seven correct significant digits).

Approximation 2 (degree-4 series): Using $e^x\cos x=1+x-\dfrac{x^3}{3}-\dfrac{x^4}{6}+O(x^5)$,

$$p^*=P_4(0.01)=1+0.01-\frac{(0.01)^3}{3}-\frac{(0.01)^4}{6}=1.009999665.$$

Then

$$\big|p-p^*\big| = \big|\,1.0099996649966665-1.009999665\,\big| \approx 3.333555653739495\times 10^{-12},$$

$$\frac{\big|p-p^*\big|}{|p|}\approx 3.30055124696551\times 10^{-12}.$$

Interpretation: relative error $\sim 10^{-12}$, a much tighter accuracy.

1.2.4 Rounding error

In finite-precision arithmetic, a real number x is replaced by a nearby representable floating–point number $\mathrm{fl}(x)$. The *rounding error* is the discrepancy

$$\text{absolute rounding error} = \big|x - \mathrm{fl}(x)\big|.$$

Under the usual IEEE 754 round-to-nearest (ties-to-even) rule, every real x is mapped to the nearest machine number, so

$$\big|x - \mathrm{fl}(x)\big| \le \tfrac{1}{2}\,\mathrm{ulp}(x),$$

where $\mathrm{ulp}(x)$ is the spacing between consecutive floating–point numbers near x. For *normalized* IEEE binary64 (double precision) numbers one also has the relative-error model

$$\mathrm{fl}(x) = x(1+\delta), \qquad |\delta| \le u, \qquad u = 2^{-53} \approx 1.11 \times 10^{-16},$$

so the *relative rounding error* is at most u (unit roundoff).

Example 3: $0.1 + 0.2$ vs 0.3 in computer

Because 0.1 and 0.2 are rounded to nearby binary64 numbers,

$$\mathrm{fl}(0.1) + \mathrm{fl}(0.2) = 0.30000000000000004440892098500626\ldots \neq 0.3,$$

so

$$\big|\mathrm{fl}(0.1+0.2) - 0.3\big| \approx 5.551115123125783 \times 10^{-17}.$$

This visible "0.30000000000000004" effect is a classic illustration of rounding error in floating–point arithmetic.

Remark. Rounding error is inevitable in finite precision. Absolute error is bounded by $\frac{1}{2}$ulp, and relative error is bounded by the unit roundoff u. These small errors can accumulate or amplify in algorithms, which is why **numerical stability** matters.

Example 4: Two equivalent formulas for the x–intercept

Given two points (x_0, y_0) and (x_1, y_1) with $y_1 \neq y_0$, the x–intercept of the line through them can be written as

$$x = \frac{x_0 y_1 - x_1 y_0}{y_1 - y_0} \qquad \text{or} \qquad x = x_0 - \frac{(x_1 - x_0)\, y_0}{y_1 - y_0}.$$

They are algebraically identical, since

$$x_0 - \frac{(x_1 - x_0)\, y_0}{y_1 - y_0} = \frac{x_0(y_1 - y_0) - (x_1 - x_0)y_0}{y_1 - y_0} = \frac{x_0 y_1 - x_1 y_0}{y_1 - y_0}.$$

Rounding error (Finite–precision effects):

Lets perform the 3 digit rounding by taking $(x_0, y_0) = (1.31,\ 3.24)$, $(x_1, y_1) = (1.93,\ 4.76)$, and also round after each operation to 3 significant digits.

Formula (1): $x = \dfrac{x_0 y_1 - x_1 y_0}{y_1 - y_0}$

$$\begin{aligned}
x_0 y_1 &= 1.31 \times 4.76 = 6.2356 \ \rightarrow\ \underline{6.24},\\
x_1 y_0 &= 1.93 \times 3.24 = 6.2532 \ \rightarrow\ \underline{6.25},\\
\text{numerator} &= 6.24 - 6.25 = -0.01 \ \ \text{(cancellation)},\\
\text{denominator} &= 4.76 - 3.24 = 1.52 \ \ \text{(exact here)},\\
x &\approx \frac{-0.0100}{1.52} = -0.00658.
\end{aligned}$$

$\Rightarrow$ Large *subtractive cancellation* in the numerator: two nearly equal products are subtracted.

Formula (2): $x = x_0 - \dfrac{(x_1 - x_0)\, y_0}{y_1 - y_0}$

$$\begin{aligned} x_1 - x_0 &= 1.93 - 1.31 = 0.62, \\ (0.62)\, y_0 &= 0.62 \times 3.24 = 2.0088 \;\to\; \underline{2.01}, \\ \frac{2.01}{1.52} &= 1.322\ldots \;\to\; \underline{1.32}, \\ x &\approx 1.31 - 1.32 = -0.0100. \end{aligned}$$

$\Rightarrow$ Less cancellation here; result closer to the true value $x_{\text{true}} = x_0 - \dfrac{(x_1 - x_0)y_0}{y_1 - y_0} = -0.011578947\ldots$. (So $|x - x_{\text{true}}|/|x_{\text{true}}| \approx 13.6\%$ for formula (2) vs. $\approx 43.2\%$ for formula (1).)

Which formula is numerically better? it depends....

Both formulas share the same denominator $y_1 - y_0$; if $|y_1 - y_0|$ is tiny (nearly horizontal line), *any* method is ill-conditioned. The difference is in the *numerators* you actually compute:

- **Formula (1)** uses $x_0 y_1 - x_1 y_0$. It suffers when the products are nearly equal, i.e. when
$$x_0 y_1 \approx x_1 y_0 \quad \Longleftrightarrow \quad \frac{y_0}{x_0} \approx \frac{y_1}{x_1},$$
meaning the two points are nearly proportional (the line passes near the origin). *Then* subtracting two close large numbers causes severe cancellation (as in the example), so avoid formula (1).

- **Formula (2)** uses the difference $x_1 - x_0$. It suffers when $x_1 \approx x_0$ (nearly vertical line), because the subtraction $x_1 - x_0$ loses significant digits before being scaled by $y_0/(y_1 - y_0)$. *Then* formula (1) can be safer (provided $x_0 y_1$ and $x_1 y_0$ are not nearly equal).

Rules to follow: Choose the form that avoids subtracting *nearly equal* quantities. Concretely,

$$\begin{cases} \text{Prefer } x = x_0 - \dfrac{(x_1 - x_0)y_0}{y_1 - y_0}, & \text{if } |x_0 y_1 - x_1 y_0| \ll \max(|x_0 y_1|, |x_1 y_0|), \\ \text{Prefer } x = \dfrac{x_0 y_1 - x_1 y_0}{y_1 - y_0}, & \text{if } |x_1 - x_0| \ll \max(|x_0|, |x_1|). \end{cases}$$

1.2.5 Error propagation

Let $y = f(x)$ with f differentiable at x, and suppose $x \neq 0$, $f(x) \neq 0$. For a small perturbation Δx, define $\Delta y = f(x + \Delta x) - f(x)$.

Linearization / first-order Taylor:

$$f(x + \Delta x) = f(x) + f'(x)\, \Delta x + \underbrace{\tfrac{1}{2} f''(\xi)\, (\Delta x)^2}_{\text{higher order}}, \quad \xi \text{ between } x \text{ and } x + \Delta x.$$

Hence, for small $|\Delta x|$,

$$\Delta y \approx f'(x)\,\Delta x.$$

Divide by $|y| = |f(x)|$ and multiply by $\dfrac{|x|}{|x|}$:

$$\frac{|\Delta y|}{|y|} \;\approx\; \frac{|f'(x)\,\Delta x|}{|f(x)|} = \Big|\frac{x f'(x)}{f(x)}\Big|\;\frac{|\Delta x|}{|x|} = \kappa_f(x)\frac{|\Delta x|}{|x|}.$$

The neglected term is $O\big(\Delta x^2\big)$, so the approximation is valid to first order.

Condition number

$$\kappa_f(x) \;:=\; \Big|\frac{x f'(x)}{f(x)}\Big| \;=\; \Big|\frac{d\ln f(x)}{d\ln x}\Big|.$$

This is refered to the *condition number* of f at x: it measures how much a small relative change in x is amplified (to first order) in the relative change of $y = f(x)$.

Example 5: A subtractive cancellation example. Consider $y = f(a, b) = a - b$, let's see how small errors in a and b propagate to the relative error in y. For small perturbations $\Delta a, \Delta b$, the change in y is:

$$\Delta y = (a + \Delta a) - (b + \Delta b) - (a - b) = \Delta a - \Delta b.$$

Taking absolute values:

$$|\Delta y| \le |\Delta a| + |\Delta b|.$$

Relative error in y is approximately:

$$\frac{|\Delta y|}{|y|} \lesssim \frac{|\Delta a| + |\Delta b|}{|a - b|}.$$

If we normalize input errors relative to a, b, i.e. $|\Delta a| \approx \epsilon|a|, |\Delta b| \approx \epsilon|b|$ for some small rounding error ϵ, then:

$$\frac{|\Delta y|}{|y|} \approx \frac{\epsilon|a| + \epsilon|b|}{|a - b|} = \epsilon\frac{|a| + |b|}{|a - b|}.$$

If $a \approx b$, the denominator is small, so relative error can blow up even if $|\Delta a|, |\Delta b|$ are tiny (rounding-level). For example, choose $a = 1.2345,\ b = 1.2335 \Rightarrow y = a - b = 0.0010$. Suppose inputs are rounded to 3 significant digits: $a^* = 1.23,\ b^* = 1.23$, giving $y^* = 0.00$. Absolute error $= 10^{-3}$, relative error $= 100\%$. Here $\kappa_{\text{rel}} \approx \dfrac{|a| + |b|}{|a - b|} \approx \dfrac{2.468}{0.001} \approx 2468$, so small input relative errors are massively amplified.

Condition number for matrix:

The concept of condition number for matrices generalizes the idea you saw with subtractive cancellation. It measures how sensitive the solution of a linear system $Ax = b$ is to small changes in the data A and b.

For a nonsingular matrix A, the (relative) condition number with respect to inversion (or solving $Ax = b$) is defined as:

$$\kappa(A) = \|A\| \cdot \big\|A^{-1}\big\|$$

where $\|\cdot\|$ is some consistent matrix norm (commonly the 2 norm or spectral norm). $\|A\|$ measures the "size" of A. $\|A^{-1}\|$ measures how "large" the inverse is (how sensitive the solution is to perturbations).

For a matrix $A \in \mathbb{R}^{m\times n}$, the 2-norm (spectral norm) is defined as

$$\|A\|_2 = \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2}.$$

which corresponds to the maximum "stretch factor" (maximum eigenvalue) that A applies to a vector in the Euclidean norm.

2 Interpolation and Approximation

2.1 Lagrange interpolation, Newton interpolation, and divided differences

2.1.1 Lagrange Interpolation

Given distinct nodes $x_0, x_1, \ldots, x_n$ with data $y_k = f(x_k)$, the Lagrange basis polynomials are defined as follows:

$$L_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}, \qquad k = 0, 1, \ldots, n,$$

(Take a product over the index j starting from 0 up to n, and exclude the term where $j = k$). The unique interpolating polynomial $p_n \in \mathbb{P}_n$ satisfying $p_n(x_k) = y_k$ is:

$$p_n(x) = \sum_{k=0}^{n} y_k \, L_k(x).$$

Where does the formula come from?

We want a polynomial $p_n \in \mathbb{P}_n$ such that $p_n(x_k) = y_k$ at distinct nodes $x_0, \ldots, x_n$. Construct basis polynomials L_k that are 1 at x_k and 0 elsewhere:

$$L_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j} \quad \Longrightarrow \quad L_k(x_i) = \delta_{ik}.$$

Then we have:

$$p_n(x) = \sum_{k=0}^{n} y_k \, L_k(x).$$

Uniqueness (why there is only one)

Prove: If $q_n \in \mathbb{P}_n$ also satisfies $q_n(x_k) = y_k$, then $r := p_n - q_n \in \mathbb{P}_n$ has the $n+1$ distinct roots $x_0, \ldots, x_n$. A nonzero polynomial of degree $\leq n$ cannot have $n+1$ distinct roots, hence $r \equiv 0$ and $p_n = q_n$.

Example 1 (linear, $n = 1$)

Nodes $x_0 = 0$, $x_1 = 1$; data $y_0 = 1$, $y_1 = e$ (sample points of $f(x) = e^x$).

Then we have:

$$L_0(x) = 1 - x, \quad L_1(x) = x, \qquad p_1(x) = y_0 L_0(x) + y_1 L_1(x) = 1 + x(e - 1).$$

Evaluate $p_1(x = 0.2)$: $p_1(0.2) = 1 + 0.2\,(e - 1)$. (a good local approximation to $e^{0.2}$).

Example 2 (quadratic, $n = 2$)

Given sample nodes:

$$(x_1, y_1) = (-1, 0), \quad (x_2, y_2) = (0, 1), \quad (x_3, y_3) = (1, 0)$$

By definition, we have:

$$L_j(x) = \prod_{\substack{m=1 \\ m \neq j}}^{3} \frac{x - x_m}{x_j - x_m}, \quad \text{so that } L_j(x_i) = \delta_{ij}$$

Compute each L_j:

1. For $j = 1$ (node -1):

$$L_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} = \frac{(x - 0)(x - 1)}{(-1 - 0)(-1 - 1)} = \frac{x(x - 1)}{(-1)(-2)} = \frac{x(x - 1)}{2}.$$

2. For $j = 2$ (node 0):

$$L_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} = \frac{(x + 1)(x - 1)}{(0 + 1)(0 - 1)} = \frac{(x + 1)(x - 1)}{1 \cdot (-1)} = -(x+1)(x-1) = 1-x^2.$$

3. For $j = 3$ (node 1):

$$L_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} = \frac{(x + 1)(x - 0)}{(1 + 1)(1 - 0)} = \frac{(x + 1)x}{2}.$$

Interpolating polynomial with data $y_1 = 0, y_2 = 1, y_3 = 0$, the lagrangian interpolant is:

$$p_2(x) = y_1 L_1(x) + y_2 L_2(x) + y_3 L_3(x) = L_2(x) = 1 - x^2,$$

which exactly interpolates the three points.

Existence and uniqueness of Lagrange Interpolation

Given distinct nodes $x_0, \ldots, x_n \in \mathbb{R}$ and data values $y_0, \ldots, y_n \in \mathbb{R}$, there exists a unique polynomial p of degree at most n such that $p(x_i) = y_i$ for $i = 0, \ldots, n$.

Proof. Uniqueness has been shown earlier.

For existence, define the Lagrange basis polynomials

$$L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \quad i = 0, \ldots, n,$$

which satisfy $L_i(x_j) = \delta_{ij}$ and $\deg L_i \leq n$. Then the polynomial

$$p(x) := \sum_{i=0}^{n} y_i L_i(x)$$

satisfies $p(x_j) = y_j$ for every j and $\deg p \leq n$. Thus such a polynomial exists. □

Remark: The prove above it based on the Lagrange form of the interpolating polynomial (A formula-based existence proof). For more general case, we can assume a general polynomial at order of at most n:

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

and show via linear algebra that there is exactly one set of coefficients $(a_0, \ldots, a_n)$ that satisfies.

Since

$$p(x_i) = y_i \quad \text{for } i = 0, 1, \ldots, n,$$

where the data points ($x_i, y_i = f(x_i)$) are given, and all x_i are distinct. Substituting into the formula for $p(x)$, we get:

$$a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n = y_i,$$

which is a linear equation with the unknowns of $a_0, \ldots, a_n$. For all $n+1$ equations, we have:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Note that this matrix is called the Vandermonde matrix V, and its determinant is $\det(V) = \prod_{0 \le i < j \le n} (x_j - x_i) \neq 0$. That means the linear system has exactly one solution for the coefficients.

Can you show both polynomials are equivalent?

Let $p_1(x) = \sum_{k=0}^{n} a_k x^k$ be the solution of $\mathrm{Va} = y$. Let $p_2(x) = \sum_{i=0}^{n} y_i L_i(x)$ be the Lagrange polynomial. Both satisfy $p_1(x_j) = p_2(x_j) = y_j$ for all n points. Since an interpolant of degree $\le n$ is unique, $p_1 \equiv p_2$.

Barycentric Interpolation Formula:

The barycentric form rewrites the interpolant as:

$$p(x) = \frac{\sum_{j=0}^{n} \frac{w_j}{x - x_j} y_j}{\sum_{j=0}^{n} \frac{w_j}{x - x_j}}, \quad x \neq x_j$$

The weights w_j depend only on the nodes x_j, not on y_j. For distinct nodes $x_0, \ldots, x_n$:

$$w_j = \frac{1}{\prod_{\substack{k=0 \\ k \neq j}}^{n} (x_j - x_k)}.$$

The evaluation of $p(x)$ for many x-values is efficient: $O(n)$ per evaluation, compared to $O(n^2)$.

Example 1: Code implementation of Lagrange interpolation over 5 points

Listing 1: MATLAB: Lagrange interpolant and basis plots

```
%% Lagrange interpolation over 5 points and basis plots
x = [0 1 2 3 4]; % nodes (distinct)
y = [1 3 2 5 4]; % data values

n = numel(x); % Return points count

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%part 1%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Compute barycentric weights
w = ones(1,n);
```

```
for i = 1:n
   for j = 1:n
      if j ~= i
         w(i) = w(i) / (x(i) - x(j));
      end
   end
end

% Evaluation grid, add +-0.2 to the ending points
xx = linspace(min(x)-0.2, max(x)+0.2, 1000);

% Evaluate interpolant using barycentric formula
p = zeros(size(xx));
for k = 1:numel(xx)
   z = xx(k);
   idx = find(abs(z - x) < 1e-14, 1);
   if ~isempty(idx)
      p(k) = y(idx);
   else
      numerator = sum( w .* (y ./ (z - x)) );
      denomenator = sum( w ./ (z - x) );
      p(k) = numerator / denomenator;
   end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%part 2%%%%%%%%%%%%%%%%%

% Compute each Lagrange basis L_i(x) explicitly for later plotting
Li = zeros(n, numel(xx));
for i = 1:n
   % L_i(z) = product_{j != i} (z - x_j)/(x_i - x_j)
   denom = prod(x(i) - x([1:i-1, i+1:n]));
   for k = 1:numel(xx)
      z = xx(k);
      Li(i,k) = prod(z - x([1:i-1, i+1:n])) / denom;
   end
end

% Plot interpolant and basis
figure;
subplot(2,1,1);
plot(xx, p, 'k-', 'LineWidth', 1.8); hold on;
plot(x, y, 'ro', 'MarkerFaceColor', 'r');
grid on; box on;
xlabel('x'); ylabel('p(x)');
title('Lagrange Interpolant through 5 Points');
legend('Interpolant','Data points','Location','best');

subplot(2,1,2);
hold on; grid on; box on;
```

```
colors = lines(n);
for i = 1:n
   plot(xx, Li(i,:), 'Color', colors(i,:), 'LineWidth', 1.5);
end
for i = 1:n
   plot(x, double(x==x(i)), 'o', 'Color', colors(i,:), 'MarkerFaceColor', colors
       (i,:)); % marks basis=1 at x_i
end
xlabel('x'); ylabel('L_i(x)');
title('Lagrange Basis Functions');
legend(arrayfun(@(i) sprintf('L_{%d}(x)',i-1),1:n,'UniformOutput',false),'
    Location','best');
```

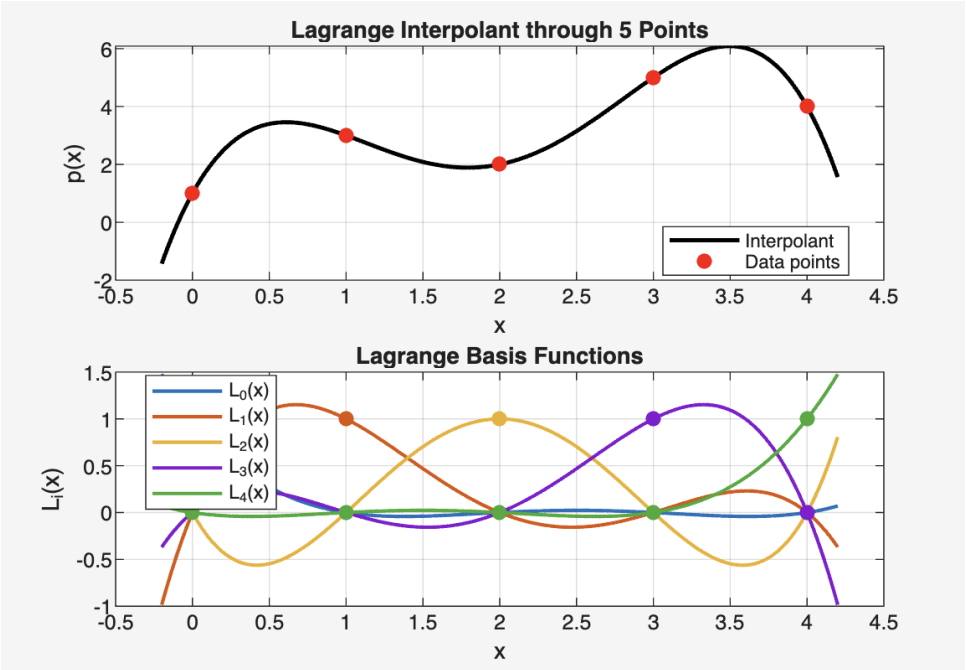

Figure 1: Lagrange interpolating polynomial and basis functions.

***Some analysis on the error term (how good is the lagrangian interpolation?)**

Let $x_0, \ldots, x_n$ be distinct nodes in $[a, b]$, and let

$$P_n(x) = \sum_{i=0}^{n} f(x_i)\, \ell_i(x), \quad \ell_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

be the Lagrange interpolant of $f \in C^{n+1}[a, b]$. Write the error as $R_n(x) = f(x) - P_n(x)$.

Next, Lets define a nodal polynomial as follows:

$$W(t) = \prod_{i=0}^{n} (t - x_i)$$

and an auxiliary function:

$$\psi(t) = (f(t) - P_n(t)) - \frac{W(t)}{W(x)} (f(x) - P_n(x)) = R_n(t) - \frac{W(t)}{W(x)} R_n(x).$$

Noticing that, the above auxiliary function satisfies that $\psi(x_i) = 0$ for all i (since $R_n(x_i) = 0$), and also $\psi(x) = 0$. Hence the auxiliary $\psi(t)$ has a total of $n+2$ distinct zeros. Which means that by applying Generalized Rolle's theorem (on page 8), there must exist some $\xi \in (a, b)$ with

$$\psi^{(n+1)}(\xi) = 0.$$

Differentiate $\psi(t) = R_n(t) - \frac{W(t)}{W(x)} R_n(x)$ by $(n+1)$ times with respect to t. Note that terms depend on x are treated as constants ($R_n(x)$ and $W(x)$), we have:

$$\psi^{(n+1)}(t) = R_n^{(n+1)}(t) - \frac{R_n(x)}{W(x)} W^{(n+1)}(t)$$

Regarding the $1st$ term on the RHS, we have $R_n^{(n+1)}(t) = f^{(n+1)}(t) - P_n^{(n+1)}(t) = f^{(n+1)}(t)$, since P_n has degree $\leq n$ and thus $P_n^{(n+1)} \equiv 0$. For the $2nd$ term, W is monic of degree $n+1$, and the $n+1st$ derivative of a monic of degree $n+1$ polynomial gives $(n+1)!$. Therefore, we have:

$$\psi^{(n+1)}(t) = f^{(n+1)}(t) - \frac{R_n(x)}{W(x)} (n+1)!$$

Given:

$$0 = \psi^{(n+1)}(\xi) = f^{(n+1)}(\xi) - \frac{(n+1)!}{W(x)} R_n(x),$$

which derives:

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

which is the classical pointwise remainder formula.

Mean Value Theorem (Or often referred as Cauchy form of the reminder.) If $f \in C^{n+1}[a, b]$, then for each $x \in [a, b]$ there exists $\xi = \xi(x) \in (a, b)$ with

$$f(x) - p_n(x) = R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} W_{n+1}(x), \qquad W_{n+1}(x) := \prod_{k=0}^{n} (x - x_k).$$

Noticing that the factor $\frac{f^{(n+1)}(\xi)}{(n+1)!}$ captures the dependence on the higher derivative of f, which ξ's existence is guarantted by Roll's theorem, while the factor $\prod_{i=0}^{n} (x - x_i)$ depends only on the interpolation nodes and the evaluation point. Thus the node choice (which controls $|W_{n+1}(x)|$) strongly affects the size of the error.

For some $\xi = \xi(x)$ between the smallest and largest node. Taking absolute values and a bound $M := \max_{t \in [a,b]} |f^{(n+1)}(t)|$ on the interval of interest (symbol ":=" read as "is defined as"),

$$|f(x) - p_n(x)| \leq \frac{M}{(n+1)!} |W_{n+1}(x)|.$$

The function $f(X)$ fixes M, but the nodes $\{x_k\}$ entirely determine ω_{n+1}. Hence, for a given $f(X)$, $|W_{n+1}(x)|$ is the factor you can control by choosing the nodes; smaller $|W_{n+1}(x)|$ means smaller interpolation error.

With equally spaced nodes, $W_{n+1}(x)$ typically grows very large near the endpoints. This magnifies the error bound and explains the oscillations (Runge phenomenon) seen for high-degree interpolation on uniform grids.

Example 3: Choose $x_0 = 0$, $x_1 = \frac{\pi}{4}$, $x_2 = \frac{\pi}{2}$ with $y_0 = 0$, $y_1 = \frac{\sqrt{2}}{2}$, $y_2 = 1$.

Then $p_2(x) = \sum_{k=0}^{2} y_k L_k(x)$. Let's check the error at $x = \frac{\pi}{6}$,

$$p_2\Big(\tfrac{\pi}{6}\Big) \approx 0.517428, \qquad \sin\Big(\tfrac{\pi}{6}\Big) = 0.5, \qquad \text{error } \approx 1.7428 \times 10^{-2}.$$

An error bound follows from

$$|f(x) - p_2(x)| = \left|\frac{f^{(3)}(\xi)}{3!}(x - x_0)(x - x_1)(x - x_2)\right| \le \frac{1}{6}\left|(x-0)\Big(x - \tfrac{\pi}{4}\Big)\Big(x - \tfrac{\pi}{2}\Big)\right|,$$

since $|f^{(3)}(t)| = |-\cos t| \le 1$.

Choosing nodes (Runge vs. Chebyshev)

The error factor splits into a part from the function, $\frac{f^{(n+1)}(\xi)}{(n+1)!}$, and a pure *node geometry* part, $\omega_{n+1}(x)$. By choosing nodes that keep $|\omega_{n+1}(x)|$ small (e.g., Chebyshev nodes), you directly and significantly improve the interpolation accuracy.

Equally spaced nodes with high degree may cause endpoint oscillations (*Runge phenomenon*). For 1D problem, on the domain of $[a, b]$, the Chebyshev nodes is given as:

$$x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\Big(\frac{2k+1}{2(n+1)}\pi\Big), \quad k = 0, \ldots, n,$$

minimize $\max_{x\in[a,b]} |\omega_{n+1}(x)|$ and typically yield smaller uniform errors.

The Chebyshev nodes are a special set of interpolation points chosen to reduce oscillations (the Runge phenomenon) when interpolating on an interval, usually $[-1, 1]$. These are the roots of the Chebyshev polynomial $T_n(x)$ and exclude the endpoints:

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

The reason the cosine appears is that the Chebyshev polynomial $T_n(x) = \cos\big(n \arccos x\big)$ oscillates uniformly when expressed in terms of the angle $\theta = \arccos x$. **Its extrema and roots are equally spaced in this angle variable rather than in the x-coordinate.** By mapping those equally spaced angles back to the x-domain using the cosine function, the resulting nodes lie in $[-1, 1]$ and become naturally clustered near the endpoints, which helps control the large oscillations seen with equally spaced nodes.

Someting about the Chebyshev polynomials

The Chebyshev polynomials of the first kind are defined by

$$T_n(x) = \cos\big(n \arccos x\big), \quad x \in [-1, 1].$$

They satisfy the recurrence

$$T_{n+1}(x) = 2x\,T_n(x) - T_{n-1}(x),$$

with $T_0(x) = 1$, $T_1(x) = x$.

First Few Chebyshev Polynomials Using the above definition, or equivalently a recurrence relation, we get:

$$
\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2x^2 - 1 \\
T_3(x) &= 4x^3 - 3x \\
T_4(x) &= 8x^4 - 8x^2 + 1, \\
&\cdots
\end{aligned}
$$

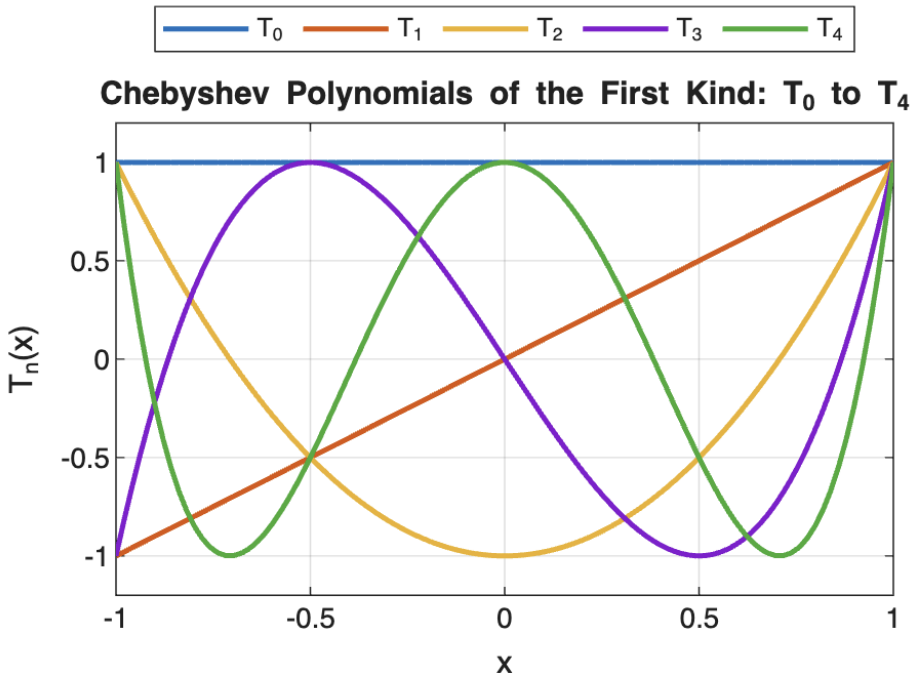

Figure 2: Chebyshev polynomial

2.1.2 Newton Interpolation

(It is just an alternative representations of Lagrangian polynomial!)

Divided differences and Newton form

Given distinct nodes $x_0, \ldots, x_n$ and data $y_k = f(x_k)$, define the divided differences recursively:

$$
\begin{aligned}
&f[x_k] = y_k, \\
&f[x_i, x_{i+1}, \ldots, x_{i+m}] = \frac{f[x_{i+1}, \ldots, x_{i+m}] - f[x_i, \ldots, x_{i+m-1}]}{x_{i+m} - x_i} \quad (m \geq 1).
\end{aligned}
$$

Then, the Newton interpolating polynomial is:

$$
p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \cdots + f[x_0, \ldots, x_n] \prod_{k=0}^{n-1} (x - x_k).
$$

It can be evaluated efficiently via the nested form:

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

where $a_m = f[x_0, \ldots, x_m]$.

Divided-difference table template

x_k	$f[x_k]$	$f[x_k, x_{k+1}]$	$f[x_k, x_{k+1}, x_{k+2}]$	$\cdots$
x_0	y_0	$\dfrac{y_1 - y_0}{x_1 - x_0}$	$\dfrac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$	$\cdots$
x_1	y_1	$\dfrac{y_2 - y_1}{x_2 - x_1}$	$\vdots$	
x_2	y_2	$\vdots$		
$\vdots$	$\vdots$			

The first entries of each column $(f[x_0], f[x_0, x_1], f[x_0, x_1, x_2], \ldots)$ are the Newton coefficients $a_0, a_1, a_2, \ldots$.

Some explanation about the Newton's formula. Let

$$P_2(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1),$$

with the interpolation conditions $P_2(x_i) = f(x_i)$ for $i = 0, 1, 2$.

To determine a_0:

$$P_2(x_0) = a_0 = f(x_0) \quad \Rightarrow \quad a_0 = f(x_0).$$

To determine a_1:

$$\begin{aligned} P_2(x_1) &= a_0 + a_1(x_1 - x_0) = f(x_1) \\ a_1 &= \frac{f(x_1) - a_0}{x_1 - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = f[x_0, x_1]. \end{aligned}$$

For a_2:

$$\begin{aligned} P_2(x_2) &= a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = f(x_2), \\ a_2 &= \frac{f(x_2) - a_0 - a_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} \\ &= \frac{f(x_2) - f(x_0)}{(x_2 - x_0)(x_2 - x_1)} - \frac{a_1}{x_2 - x_1} \\ &= \frac{f(x_2) - f(x_0)}{(x_2 - x_0)(x_2 - x_1)} - \frac{f[x_0, x_1]}{x_2 - x_1}. \\ &= \frac{f(x_2) - f(x_1) + f(x_1) - f(x_0)}{(x_2 - x_0)(x_2 - x_1)} - \frac{f[x_0, x_1]}{x_2 - x_1} \\ &= \frac{f(x_2) - f(x_1)}{(x_2 - x_0)(x_2 - x_1)} + \frac{f(x_1) - f(x_0)}{(x_2 - x_0)(x_2 - x_1)} - \frac{f[x_0, x_1]}{x_2 - x_1} \\ &= \frac{f[x_1, x_2]}{x_2 - x_0} + \frac{f[x_0, x_1]}{x_2 - x_1} \cdot \frac{x_1 - x_0}{x_2 - x_0} - \frac{f[x_0, x_1]}{x_2 - x_1} \\ &= \frac{f[x_1, x_2]}{x_2 - x_0} + \frac{f[x_0, x_1]}{x_2 - x_1} \left(\frac{x_1 - x_0 - (x_2 - x_0)}{x_2 - x_0} \right) \\ &= \frac{f[x_1, x_2]}{x_2 - x_0} - \frac{f[x_0, x_1]}{x_2 - x_0} \\ &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = f[x_0, x_1, x_2]. \end{aligned}$$

Example 1: Exact recovery of a quadratic data from $f(x) = x^2$, at $x_0 = 1,\ x_1 = 2,\ x_2 = 4$

node	$f[x_k]$	$f[x_k, x_{k+1}]$	$f[x_k, x_{k+1}, x_{k+2}]$
1	1	$\frac{4-1}{2-1} = 3$	$\frac{6-3}{4-1} = 1$
2	4	$\frac{16-4}{4-2} = 6$	
4	16		

Thus $a_0 = 1,\ a_1 = 3,\ a_2 = 1$ and

$$p_2(x) = 1 + 3(x-1) + 1\,(x-1)(x-2) = x^2 \quad \text{(exact)}.$$

Error term

If $f \in C^{n+1}[a,b]$, then for each $x \in [a,b]$ there exists $\xi = \xi(x) \in (a,b)$ such that

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\,\omega_{n+1}(x), \qquad \omega_{n+1}(x) := \prod_{k=0}^{n}(x - x_k).$$

Equivalently: $f(x) - p_n(x) = f[x_0, \ldots, x_n, x]\ \omega_{n+1}(x)$.

Why Newton form is practical?

- The coefficients $a_m = f[x_0, \ldots, x_m]$ can be computed from a simple triangular divided-difference table in $O(n^2)$ time.
- **Incremental update:** If a new node x_{n+1} is added, you only compute one new column in the table and add the term $f[x_0, \ldots, x_{n+1}] \prod_{k=0}^{n}(x - x_k)$; no need to recalculate all the entries in the table from scratch again.
- Evaluation at a point is $O(n)$ via the nested form and is numerically stable when nodes are ordered.

Example 2: Approximating e^x on $[0,1]$ using Newton interpolation.

Given three nodes $x_0 = 0,\ x_1 = \frac{1}{2},\ x_2 = 1$ with

$$f[x_0] = e^0 = 1, \quad f[x_1] = e^{1/2} \approx 1.6487212707, \quad f[x_2] = e^1 \approx 2.7182818285.$$

Calculating the following differences:
First differences:

$$f[x_0, x_1] = \frac{1.6487212707 - 1}{0.5} \approx 1.2974425414002564,$$

$$f[x_1, x_2] = \frac{2.7182818285 - 1.6487212707}{0.5} \approx 2.139121115517834.$$

Second difference:

$$f[x_0, x_1, x_2] = \frac{2.139121115517834 - 1.2974425414002564}{1 - 0} \approx 0.8416785741175774.$$

Hence

$$p_2(x) = 1 + 1.2974425414002564\,x + 0.8416785741175774\,x\Big(x - \tfrac{1}{2}\Big).$$

Check the accuracy for $x = 0.3$:

$$p_2(0.3) \approx 1.3387320479730223, \qquad e^{0.3} \approx 1.3498588075760032,$$

Therefore, the absolute error is $\approx 1.11 \times 10^{-2}$. The error bound from the remainder is

$$|e^x - p_2(x)| = \left|\frac{e^\xi}{3!}\, x\Big(x - \tfrac{1}{2}\Big)(x-1)\right| \le \frac{e}{6}\,\big|0.3 \times (-0.2) \times (-0.7)\big| \approx 1.90 \times 10^{-2},$$

which encloses the actual error.

Remarks:

(1) Newton form builds the interpolant from divided differences and is easy to update with new nodes.

(2) Evaluation is fast via nesting.

(3) The same error term as Lagrange applies; node choice still controls $|\omega_{n+1}(x)|$ and thus the interpolation accuracy.

Some variations of Newton interpolation:

Backward differences

For nodes x_2, x_1, x_0, the quadratic interpolant is

$$P_2(x) = b_0 + b_1(x - x_2) + b_2(x - x_2)(x - x_1),$$

with coefficients

$$b_0 = f(x_2), \qquad b_1 = \frac{f(x_1) - f(x_2)}{x_1 - x_2} = f[x_2, x_1], \qquad b_2 = \frac{f[x_1, x_0] - f[x_2, x_1]}{x_0 - x_2} = f[x_2, x_1, x_0].$$

Centered differences (three points)

For nodes x_0, x_1, x_2,

$$P_2(x) = C_0 + C_1(x - x_1) + \frac{C_2}{2}\Big((x - x_1)(x - x_0) + (x - x_1)(x - x_2)\Big),$$

where

$$C_0 = f(x_1), \qquad C_1 = \tfrac{1}{2}\big(f[x_0, x_1] + f[x_1, x_2]\big), \qquad C_2 = f[x_0, x_1, x_2].$$

Centered differences (even number of points)

With four nodes x_0, x_1, x_2, x_3,

$$P_3(x) = \; d_0 + d_1(x - x_1) + \frac{d_2}{2}\Big((x - x_1)(x - x_0) + (x - x_1)(x - x_2)\Big) + d_3(x - x_1)(x - x_2)(x - x_3),$$

and

$$d_0 = f(x_1).$$

Example 3: Code implementation of forward Newton Interpolation

Listing 2: MATLAB: Divided differences and Newton interpolants (forward backward)

```
%% Divided differences for e^x over 5 nodes and Newton-form evaluation

% Data and evaluation grid
f = @(x) exp(x); % target function
x0 = [0 0.3 1 1.5 2]; % given nodes (distinct, ascending)
```

```
y0 = f(x0); % function values at nodes
x = linspace(min(x0), max(x0), 2000); % evaluation points for plotting

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% part 1 %%%%%%%%%%%%%%%%

% First through fourth divided differences on the 5-point stencil
d1 = (y0(2:5) - y0(1:4)) ./ (x0(2:5) - x0(1:4)); % 4 values
d2 = (d1(2:4) - d1(1:3)) ./ (x0(3:5) - x0(1:3)); % 3 values
d3 = (d2(2:3) - d2(1:2)) ./ (x0(4:5) - x0(1:2)); % 2 values
d4 = (d3(2) - d3(1)) ./ (x0(5) - x0(1)); % scalar

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% part 2 %%%%%%%%%%%%%%%%

% Construct Newton's interpolant (forward form, based at x0(1))
X1 = (x - x0(1));
X2 = X1 .* (x - x0(2));
X3 = X2 .* (x - x0(3));
X4 = X3 .* (x - x0(4));
y_forw = y0(1) + d1(1)*X1 + d2(1)*X2 + d3(1)*X3 + d4*X4;

% Construct Newton's interpolant (backward form, based at x0(5))
X1b = (x - x0(5));
X2b = X1b .* (x - x0(4));
X3b = X2b .* (x - x0(3));
X4b = X3b .* (x - x0(2));
y_back = y0(5) + d1(4)*X1b + d2(3)*X2b + d3(2)*X3b + d4*X4b;

% Plot: forward, backward, and true function
figure; hold on; grid on; box on;
plot(x, y_forw, 'LineWidth', 1.8);
plot(x, y_back, 'LineWidth', 1.8);
plot(x, f(x), '--', 'LineWidth', 1.2);
plot(x0, y0, 'ko', 'MarkerFaceColor', 'k'); % nodes
xlabel('x'); ylabel('y');
title('Newton Interpolation via Divided Differences');
legend('Forward Newton','Backward Newton','f(x)=e^x','Nodes','Location','best');
```

For practice, think about implementing the backward and centered difference schemes, and extend them to the general "n" point case.

2.1.3 Divided differences vs. Newton interpolation

Divided differences are the coefficients/recurrences computed from the data $\{(x_k, y_k)\}$. *Newton interpolation* is the polynomial written in Newton's nested form whose coefficients are exactly those divided differences:

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

So: divided differences are the tool/data structure; Newton interpolation is the resulting polynomial representation.

Divided differences (definition & properties). Given distinct nodes $x_0, \dots, x_n$ with $f[x_k] = y_k$,

$$f[x_i, x_{i+1}, \dots, x_{i+m}] = \frac{f[x_{i+1}, \dots, x_{i+m}] - f[x_i, \dots, x_{i+m-1}]}{x_{i+m} - x_i}, \quad m \geq 1.$$

Key facts:

1. Symmetry: $f[x_{i_0}, \dots, x_{i_m}]$ is invariant under permuting the nodes.
2. Newton coefficients: $a_m = f[x_0, \dots, x_m]$ are the coefficients in the Newton form.
3. Efficient updates: adding a new node x_{n+1} requires computing one new column only.
4. Equally spaced nodes: divided differences reduce to scaled finite differences, e.g. $f[x_0, x_1] = \Delta f_0/h$, $f[x_0, x_1, x_2] = \Delta^2 f_0/(2!h^2)$.

Newton interpolation (form & evaluation). With $a_m = f[x_0, \dots, x_m]$,

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

which evaluates in $O(n)$ flops via nesting (Horner-like). The error is the same as in Lagrange form:

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\,\omega_{n+1}(x), \qquad \omega_{n+1}(x) = \prod_{k=0}^{n}(x - x_k).$$

***Beyond Newton interpolation: what divided differences can do**

1. Hermite (confluent) interpolation with derivative data. Divided differences extend naturally to *repeated nodes* to incorporate derivatives:

$$f[x_0, x_0] = f'(x_0), \quad f[x_0, x_0, x_0] = \tfrac{1}{2}f''(x_0), \ \dots$$

Thus mixed data like $f(x_0), f'(x_0), f''(x_0), f(1), f'(1), ...$ are interpolated by the Newton form using *confluent* divided differences.

Idea: When a node is repeated, the usual divided-difference quotient would divide by zero. Namely,

$$f\,[x_i, x_j] = \frac{f\,(x_j) - f\,(x_i)}{x_j - x_i}, \quad i \neq j.$$

In Hermite interpolation we define those entries by a limiting process, which turns them into derivatives. For a smooth f,

$$\boxed{f[\underbrace{x_0, \dots, x_0}_{m+1 \text{ times}}] = \frac{f^{(m)}(x_0)}{m!}}$$

and mixed entries with some coincident endpoints are computed by the standard recursion, using these derivative-based base cases.

Definition by limit (why derivatives appear):

For a repeated node x_0,

$$f[x_0, x_0] \;=\; \lim_{x\to x_0} \frac{f[x]-f[x_0]}{x-x_0} = \lim_{x\to x_0} \frac{f(x)-f(x_0)}{x-x_0} = f'(x_0).$$

Similarly,

$$\begin{aligned} f[x_0,x_0,x_0] &= \lim_{x\to x_0} \frac{f[x_0,x]-f[x_0,x_0]}{x-x_0} = \lim_{x\to x_0} \frac{\frac{f(x)-f(x_0)}{x-x_0} - f'(x_0)}{x-x_0} \\ &= \lim_{x\to x_0} \frac{f(x)-f(x_0)-f'(x_0)(x-x_0)}{(x-x_0)^2} = \frac{f''(x_0)}{2!}, \end{aligned}$$

etc...

2. Unequally spaced finite differences & derivative formulas.

For scattered interpolation nodes $x_0, \ldots, x_n$, one can express the m-th derivative of f by:

$$f^{(m)}(x_0) \;\approx\; \sum_{k=0}^{n} w_k^{(m)} f(x_k),$$

with weights $w_k^{(m)}$ obtained from divided differences.

From Lagrangian interpolation, we can write:

$$f(x) \approx p(x) = \sum_{k=0}^{n} l_k(x) f(x_k).$$

By taking the $n-th$ derivative, and evaluate at point x_0, we obtain:

$$f^{(m)}(x_0) \approx p^{(m)}(x_0) = \sum_{k=0}^{n} \ell_k^{(m)}(x_0)\, f(x_k),$$

Namely, the weight $w_k^{(m)} = \ell_k^{(m)}(x_0)$.

Example 1: using nodes x_0, x_0+h, x_0+2h, derive the finite difference scheme

$$f'(x_0) \approx \frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h},$$

via divided differences.

Let

$$x_0, \quad x_1 = x_0+h, \quad x_2 = x_0+2h,$$

The corresponding Lagrange bases are:

$$\begin{aligned} \ell_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{(x-(x_0+h))(x-(x_0+2h))}{(-h)(-2h)} \\ \ell_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = \frac{(x-x_0)(x-(x_0+2h))}{h(-h)} \\ \ell_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{(x-x_0)(x-(x_0+h))}{2h(h)} \end{aligned}$$

Differentiate the above bases at x_0, we obtain:

$\ell_0'(x_0) = -3/(2h)$, $\ell_1'(x_0) = 2/h$, and $\ell_2'(x_0) = -1/(2h)$.

So the weights $w_k^{(1)}$ are:

$$w_0^{(1)} = -\frac{3}{2h}, \quad w_1^{(1)} = \frac{4}{2h}, \quad w_2^{(1)} = -\frac{1}{2h}.$$

3. Error analysis and mean–value characterization.

The remainder can be written in two equivalent ways:

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\,\omega_{n+1}(x) \;=\; f[x_0, \ldots, x_n, x]\;\omega_{n+1}(x),$$

so the $(n+1)$-st divided difference $f[x_0, \ldots, x_n, x]$ plays the role of a generalized derivative (a mean–value form). This aids in deriving *a priori* error bounds and sensitivity estimates. For Details, see Burden's book Theorem 3.6.

4. Data shape diagnostics (monotonicity/convexity).

First and second divided differences act as discrete analogues of f' and f'':

$$\text{monotone increasing data} \iff f[x_k, x_{k+1}] \geq 0,$$

$$\text{convex data} \iff f[x_k, x_{k+1}, x_{k+2}] \geq 0.$$

Remark. "Divided differences" are more than a way to write the Newton interpolant: they are a unifying tool for interpolation (including Hermite), derivative approximation on arbitrary grids, error/shape analysis, stable piecewise representations (splines), and even advanced topics like quadrature, multistep ODE methods, and matrix function theory..

2.2 Hermite interpolation; spline interpolation

Osculating polynomial:

Let $x_0, x_1, \ldots, x_n$ be $n+1$ distinct numbers in $[a, b]$ and, for $j = 0, 1, \ldots, n$ let m_i be a nonnegative integer. Suppose that $f \in C^m[a, b]$, where $m = \max_{0 \leq i \leq n} m_j$.

The osculating polynomial approximating f is the polynomial $P(x)$ of least degree such that:

$$\frac{d^r P(x_j)}{dx^r} = \frac{d^r f(x_j)}{dx^r}, \qquad \text{for each } j = 0, 1, \ldots, n \quad \text{and} \quad r = 0, 1, \ldots, m_j.$$

1) Note that when $n = 0$ (one point), the osculating polynomial approximating f is the m_0 th Taylor polynomial for f at x_0.

2) When $m_j = 0$ for each j, the osculating polynomial is the nth Lagrange polynomial interpolating f on $x_0, x_1, \ldots, x_n$. (Only use $f(x_i)$)

3) When $m_j > 0$, the data points contains repeated nodes, we can define the confluent divided differences using corresponding derivatives, as we did on page 29.

4) When $m_j = 1$, for each $j = 0, 1, \ldots, n$, we use $(f(x_j), f'(x_j))$ (multiplicity data), gives the **Hermite polynomials.**

Hermite Interpolation:

Let $m_j \in \mathbb{N}$ be the multiplicity of data at x_j (given $f^{(r)}(x_j)$ for $r = 0, 1, \ldots, m_j - 1$). Then there exists a **unique** polynomial $p(x)$ of degree

$$\deg p \;\leq\; \Big(\sum_{j=0}^{n} m_j\Big) - 1$$

such that

$$p^{(r)}(x_j) = f^{(r)}(x_j), \qquad r = 0, 1, \ldots, m_j - 1, \quad \text{and} j = 0, \ldots, n.$$

In another word, given the nodes' values at $x_0, \ldots, x_n$ and their derivative data (again, called multiplicity), Hermite interpolation constructs a single polynomial $p(x)$ that matches function values and derivatives at those nodes.

Proof. The proof of uniqueness can be done by showing the generalized Vandermonde system is nonsingular, similar to what we did on pages 17-18.

Given those data, we have $M = \sum_{j=0}^{n} m_j$ conditions to satisfy. A polynomial of *degp* has $p + 1 = M$ coefficients, and plugging each data (a total of M data) results a linear equation, and together we have a linear system of M equations. If the data are distinct from each other, the solution to the system exits (Existence) and is unique.

□

Classical case: matching f and f' at each node

Assume $m_j = 2$ for all j (r=1, namely known data of $f(x_j)$ and $f'(x_j)$). Set n distinct nodes $x_0, \ldots, x_n$ and define the Lagrange basis

$$L_{n,j}(x) \;=\; \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}.$$

In the following, we can define the **Hermite basis pairs**:

$$H_{n,j}(x) \;=\; \Big[1 - 2(x - x_j)L'_{n,j}(x_j)\Big]\, L_{n,j}(x)^2, \qquad \widehat{H}_{n,j}(x) \;=\; (x - x_j)\, L_{n,j}(x)^2.$$

And the **Hermite interpolant** in the closed form as:

$$\begin{aligned} p_{2n+1}(x) \;&=\; \sum_{j=0}^{n} f(x_j)\, H_{n,j}(x) \;+\; \sum_{j=0}^{n} f'(x_j)\, \widehat{H}_{n,j}(x). \\ &= \sum_{j=0}^{n} \left\{ f\left(x_j\right) \left[1 - 2\left(x - x_j\right) L'_{n,j}\left(x_j\right)\right] + f'\left(x_j\right)\left(x - x_j\right) \right\} L_{n,j}(x)^2 \end{aligned}$$

Next, our goal is to derive the general expression of $L'_{n,j}(x)$ (instead of manually compute the derivatives for each $L_{n,j}$). First of all, we notice that the denominator of the $L_{n,j}(x)$ is a constant and only the numerator depends on x. So we define the new function $Q_j(x)$:

$$Q_{n,j}(x) := \prod_{\substack{k=0 \\ k \neq j}}^{n} \left(x - x_k\right).$$

Take log on both side:

$$\log Q_{n,j}(x) = \sum_{\substack{k=0 \\ k \neq j}}^{n} \log\left(x - x_k\right).$$

Differentiate the equation above:

$$\frac{Q'_{n,j}(x)}{Q_{n,j}(x)} = \sum_{\substack{k=0 \\ k \neq j}}^{n} \frac{1}{x - x_k}.$$

Since,

$$L_{n,j}(x) = \frac{Q_{n,j}(x)}{Q_{n,j}(x_j)},$$

differentiate it, we get:

$$L'_{n,j}(x) = \frac{Q'_{n,j}(x)}{Q_{n,j}(x_j)} = \frac{Q_{n,j}(x)}{Q_{n,j}(x_j)} \sum_{k\neq j} \frac{1}{x - x_k} = L_{n,j}(x) \sum_{k\neq j} \frac{1}{x - x_k}.$$

Therefore, set $x = x_j$, we derive the expression for $L'_{n,j}(x_j)$:

$$L'_{n,j}(x_j) \;=\; \sum_{\substack{k=0 \\ k\neq j}}^{n} \frac{1}{x_j - x_k}.$$

Properties of Hermite bases (why this works?)

$$\boxed{\begin{aligned} &p_{2n+1}(x) = \sum_{j=0}^{n} f(x_j)\, H_{n,j}(x) + \sum_{j=0}^{n} f'(x_j)\, \widehat{H}_{n,j}(x). \\ &H_{n,j}(x) = \left[1 - 2(x - x_j) L'_{n,j}(x_j)\right] L_{n,j}(x)^2, \quad \widehat{H}_{n,j}(x) = (x - x_j) L_{n,j}(x)^2 \end{aligned}}$$

For all $i, j \in \{0, \ldots, n\}$,

$$H_{n,j}(x_i) = \delta_{ij}, \quad H'_{n,j}(x_i) = 0, \qquad \widehat{H}_{n,j}(x_i) = 0, \quad \widehat{H}'_{n,j}(x_i) = \delta_{ij}.$$

Hence $p(x_i) = f(x_i)$ and $p'(x_i) = f'(x_i)$.

Proof:

1. $H_{n,j}(x_i) = \delta_{ij}$.
   If $i \neq j$, then $L_j(x_i) = 0 \Rightarrow L_j(x_i)^2 = 0 \Rightarrow H_{n,j}(x_i) = 0$.
   If $i = j$, then $L_j(x_j) = 1$ and $1 - 2(x - x_j)L'_j(x_j)\big|_{x=x_j} = 1 \Rightarrow (H_{n,j}(x_j) = 1$.

2. $H'_{n,j}(x_i) = 0$.
   Rewrite $H_{n,j} = g\, L_j^2$ with $g(x) = 1 - 2(x - x_j)L'_j(x_j)$. Then

   $$H'_{n,j} = g'\, L_j^2 + g\, 2L_j L'_j, \qquad g' = -2L'_j(x_j).$$

   If $i \neq j$, L_j^2 has a *double* zero at x_i, so both terms vanish and $H'_{n,j}(x_i) = 0$.
   If $i = j$, evaluate to get

   $$H'_{n,j}(x_j) = -2L'_j(x_j) \cdot 1 + 1 \cdot 2L'_j(x_j) = 0.$$

3. $\widehat{H}_{n,j}(x_i) = 0$.
   Because $\widehat{H}_{n,j} = (x - x_j)L_j^2$, it vanishes at x_j by the factor $(x - x_j)$ and at x_i for $i \neq j$ since $L_j(x_i) = 0$.

4. $\widehat{H}'_{n,j}(x_i) = \delta_{ij}$.

   Differentiate $H_{n,j}(x) = \left[1 - 2\left(x - x_j\right) L'_{n,j}\left(x_j\right)\right] L_{n,j}(x)^2$, we have:

$$\widehat{H}'_{n,j} = L_j^2 + (x - x_j)\, 2 L_j L'_j.$$

   If $i = j$, $L_j(x_j) = 1$ and the second term is 0, so $\widehat{H}'_{n,j}(x_j) = 1$.

   If$i \neq j$, $L_j^2(x_i)$ has a double zero at x_i, hence $\widehat{H}'_{n,j}(x_i) = 0$.

To show $p'_{2n+1}(x_j) = f'(x_j)$

The idea is to differentiate $H_{n,j}(x)$ and $\widehat{H}_{n,j}(x)$ in p'_{2n+1}. And we can show that $H'_{n,j}\left(x_j\right) = 0$ and $\widehat{H}'_{n,j}\left(x_j\right) = 1$. Details can be found in Burden's book, on Page 137.

Error estimate

If f is $(2n + 2)$ times continuously differentiable on an interval containing the nodes and x, then

$$f(x) - p(x) \;=\; \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{j=0}^{n} (x - x_j)^2, \quad \text{for some } \xi \text{ between } x \text{ and } \{x_j\}.$$

The error vanishes quadratically at each node.

Proof. Define the error $E(x) := f(x) - p(x)$, and introduce an auxiliary function:

$$F(t) := E(t) - \frac{E(x)}{\omega(x)} \omega(t),$$

with $\omega(t) := \prod_{j=0}^{n} \left(t - x_j\right)^2$. Such that:

$$F\left(x_j\right) = 0, \quad F'\left(x_j\right) = 0 \quad (j = 0, \ldots, n).$$

and $F(x) = 0$. Therefore, $F(t)$ has $2(n + 1) + 1 = 2n + 3$ zeros on the interval. By generalized Rolle's theorem, there exists ξ between x and the set $\{x_j\}$ such that

$$F^{(2n+2)}(\xi) = 0.$$

Next, differentiate $F(t)$ by $2n + 2$ times:

$$F^{(2n+2)}(t) = E^{(2n+2)}(t) - \frac{E(x)}{\omega(x)} \omega^{(2n+2)}(t).$$

Since $\deg p \leq 2n+1$, we have $p^{(2n+2)} = 0$, so $E^{(2n+2)} = f^{(2n+2)}$. And $\omega^{(2n+2)}(t) = (2n+2)!$. Evaluating at $t = \xi$ gives:

$$0 = f^{(2n+2)}(\xi) - \frac{E(x)}{\omega(x)} (2n + 2)!,$$

Therefore,

$$E(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \omega(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{j=0}^{n} \left(x - x_j\right)^2.$$

Remark:

1. Is this a pointwise error or uniform error?

For each fixed x, there exists some $\xi = \xi(x)$ lying between x and the set of nodes $\{x_j\}$ such that this equality holds. The dependence of ξ on x is crucial, namely for a different x-value, you get a different ξ.

2. How do you get the uniform error bound?

$$|f(x) - p(x)| \leq \frac{\max_{\xi \in I} \left|f^{(2n+2)}(\xi)\right|}{(2n+2)!} \left|\prod_{j=0}^{n} (x - x_j)^2\right|.$$

If $f \in C^{2n+2}([a,b])$, we set:

$$M := \max_{\xi \in [a,b]} \left|f^{(2n+2)}(\xi)\right|.$$

Taking absolute values and using $\left|f^{(2n+2)}(\xi(x))\right| \leq M$ for every $x \in [a,b]]$ gives the uniform error bound:

$$\sup_{x \in [a,b]} |f(x) - p(x)| \leq \frac{M}{(2n+2)!} \sup_{x \in [a,b]} \left|\prod_{j=0}^{n} (x - x_j)\right|^2.$$

For $\left|\prod_{j=0}^{n} (x - x_j)\right|^2$, a crude bound follows, considering that $\mid x - x_j \mid \leq b - a$ for every $x \in [a,b]$,

$$\sup_{x \in [a,b]} |f(x) - p(x)| \leq \frac{M}{(2n+2)!}(b-a)^{2n+2}.$$

3. Can we do better?

Example 1: Hermite interpolation example

Use the Hermite interpolant that matches the data in Table below to approximate $f(1.5)$.

k	x_k	$f(x_k)$	$f'(x_k)$
0	1.3	0.6200860	-0.5220232
1	1.6	0.4554022	-0.5698959
2	1.9	0.2818186	-0.5811571

With nodes $x_0 = 1.3,\ x_1 = 1.6,\ x_2 = 1.9$, the quadratic Lagrange basis polynomials and their derivatives are

$$L_{2,0}(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{50}{9}x^2 - \frac{175}{9}x + \frac{152}{9}, \qquad L'_{2,0}(x) = \frac{100}{9}x - \frac{175}{9},$$

$$L_{2,1}(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = -\frac{100}{9}x^2 + \frac{320}{9}x - \frac{247}{9}, \qquad L'_{2,1}(x) = -\frac{200}{9}x + \frac{320}{9},$$

$$L_{2,2}(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{50}{9}x^2 - \frac{145}{9}x + \frac{104}{9}, \qquad L'_{2,2}(x) = \frac{100}{9}x - \frac{145}{9}.$$

The Hermite basis functions (for $j = 0, 1, 2$) are

$$H_{2,j}(x) = \left[1 - 2(x - x_j)L'_{2,j}(x_j)\right]\, L_{2,j}(x)^2,$$

$$\hat{H}_{2,j}(x) = (x - x_j)\, L_{2,j}(x)^2.$$

Plug in the nodes values, we have:

$$L'_{2,0}(x_0) = -5,$$
$$L'_{2,1}(x_1) = 0,$$
$$L'_{2,2}(x_2) = 5.$$

Therefore,

$$\begin{aligned}
H_{2,0}(x) &= [1 - 2(x - 1.3)(-5)]\, L_{2,0}(x)^2 = (10x - 12)\, L_{2,0}(x)^2,\\
H_{2,1}(x) &= [1 - 2(x - 1.6)\cdot 0]\, L_{2,1}(x)^2 = L_{2,1}(x)^2,\\
H_{2,2}(x) &= [1 - 2(x - 1.9)\cdot 5]\, L_{2,2}(x)^2 = 10(2 - x)\, L_{2,2}(x)^2,\\
\hat{H}_{2,0}(x) &= (x - 1.3)\, L_{2,0}(x)^2,\\
\hat{H}_{2,1}(x) &= (x - 1.6)\, L_{2,1}(x)^2,\\
\hat{H}_{2,2}(x) &= (x - 1.9)\, L_{2,2}(x)^2.
\end{aligned}$$

The degree-5 Hermite interpolant is:

$$p_5(x) = \sum_{j=0}^{2} f(x_j)\, H_{2,j}(x) \;-\; \sum_{j=0}^{2} f'(x_j)\, \hat{H}_{2,j}(x).$$

Evaluating at $x = 1.5$ (using the precomputed basis values),

$$\begin{aligned}
p_5(1.5) &= 0.6200860 H_{2,0}(x) + 0.4554022 H_{2,1}(x) + 0.2818186 H_{2,2}(x)\\
&\quad - 0.5220232 \hat{H}_{2,0}(x) - 0.5698959 \hat{H}_{2,1}(x) - 0.5811571 \hat{H}_{2,2}(x)\\
&= 0.6200860 \left(\frac{4}{27}\right) + 0.4554022 \left(\frac{64}{81}\right) + 0.2818186 \left(\frac{5}{81}\right)\\
&\quad - \; 0.5220232 \left(\frac{4}{405}\right) - \; 0.5698959 \left(\frac{-32}{405}\right) - \; 0.5811571 \left(\frac{-2}{405}\right)\\
&= 0.5118277,
\end{aligned}$$

so $f(1.5) \approx 0.5118277$ to the stated accuracy.

Hermite Interpolation Formulation by Divided Differences

Newton interpolant

Given distinct nodes $x_0, \ldots, x_n$, the Newton interpolant is

$$P_n(x) = f[x_0] \;+\; \sum_{k=1}^{n} f[x_0, \ldots, x_k]\, (x - x_0) \cdots (x - x_{k-1}).$$

Hermite data (values and derivatives)

Suppose we are given $f(x_i)$ and $f'(x_i)$ at $x_0, \ldots, x_n$. Define a repeated-node list

$$z_{2i} = z_{2i+1} = x_i, \qquad i = 0, \ldots, n,$$

so there are $2n + 2$ entries $z_0, \ldots, z_{2n+1}$.

(The ordering is like concatination, $f(x_0), f'(x_0), f(x_1), f'(x_1), f(x_2), f'(x_2)....$)

Divided-difference table with repeated nodes

Fill the first column by $f[z_k] = f(z_k)$. For the "undefined" first differences at repeated pairs,

$$f[z_{2i}, z_{2i+1}] \;\; f'(x_i),$$

consistent with the limit identity:

$$\lim_{\substack{y \to x \\ y \neq x}} \frac{f(y) - f(x)}{y - x} = f'(x).$$

All higher-order divided differences are then computed by the recursion as usual:

$$f[z_k, \ldots, z_{k+m}] = \frac{f[z_{k+1}, \ldots, z_{k+m}] - f[z_k, \ldots, z_{k+m-1}]}{z_{k+m} - z_k}, \qquad m \geq 1,$$

whenever the denominator is nonzero (which it is unless all endpoints coincide).

Hermite (Newton) form

With the repeated-node list $\{z_k\}$ the Hermite interpolant of degree $\leq 2n+1$ is

$$\boxed{H_{2n+1}(x) = f[z_0] \; + \; \sum_{k=1}^{2n+1} f[z_0, \ldots, z_k] \prod_{r=0}^{k-1} (x - z_r).}$$

By construction, $H_{2n+1}(x_i) = f(x_i)$ and $H'_{2n+1}(x_i) = f'(x_i)$ for all i.

Example 2: Two nodes (Generating the cubic Hermite)

Let $x_0 < x_1$, set $z_0 = z_1 = x_0$, $z_2 = z_3 = x_1$, and denote $h := x_1 - x_0$. The first three divided-difference columns are:

Table 1: Hermite divided-difference table (two nodes, repeated)

z_k	$f(z_k)$	1st divided difference	2nd divided difference	3rd divided difference
$z_0 = x_0$	$f(x_0)$	$f'(x_0)$	$\dfrac{\frac{f(x_1)-f(x_0)}{h} - f'(x_0)}{h}$	$\dfrac{\left(f'(x_1) - \frac{f(x_1)-f(x_0)}{h}\right) - \left(\frac{f(x_1)-f(x_0)}{h} - f'(x_0)\right)}{h^2}$
$z_1 = x_0$	$f(x_0)$	$\dfrac{f(x_1)-f(x_0)}{h}$	$\dfrac{f'(x_1) - \frac{f(x_1)-f(x_0)}{h}}{h}$	
$z_2 = x_1$	$f(x_1)$	$f'(x_1)$		
$z_3 = x_1$	$f(x_1)$			

Hence the cubic Hermite interpolant in Newton form is

$$\boxed{\begin{aligned} p_3(x) = f[z_0] + f[z_0, z_1]\,(x - z_0) + f[z_0, z_1, z_2]\,(x - z_0)(x - z_1) \\ + f[z_0, z_1, z_2, z_3]\,(x - z_0)(x - z_1)(x - z_2). \end{aligned}}$$

It satisfies $H_3(x_0) = f(x_0)$, $H'_3(x_0) = f'(x_0)$, $H_3(x_1) = f(x_1)$, and $H'_3(x_1) = f'(x_1)$.

***Special case: cubic Hermite on one interval** On $[x_0, x_1]$ with data $f(x_0), f'(x_0), f(x_1), f'(x_1)$ there is a unique cubic interpolant. Let $h = x_1 - x_0$ and $t = (x - x_0)/h \in [0, 1]$. Define the *cubic Hermite shape functions*

$$\begin{aligned} h_{00}(t) &= 2t^3 - 3t^2 + 1, \qquad & h_{10}(t) &= t^3 - 2t^2 + t, \\ h_{01}(t) &= -2t^3 + 3t^2, \qquad & h_{11}(t) &= t^3 - t^2. \end{aligned}$$

Then

$$\boxed{p(x) = f(x_0)\, h_{00}(t) + h\, f'(x_0)\, h_{10}(t) + f(x_1)\, h_{01}(t) + h\, f'(x_1)\, h_{11}(t).}$$

These satisfy $p(x_i) = f(x_i)$ and $p'(x_i) = f'(x_i)$ for $i = 0, 1$. $h_{ij}(t)$ are widely used as basis function (cubic spline) for FEM for beam analysis.

3 Numerical Differentiation and Quadrature

3.1 Finite difference formulas

3.1.1 Motivation of numerical differentiation and its derivation

When we don't have an explicit formula for a function but want to compute its derivatives, we can approximate them numerically. Start from the definition of 1st order derivative of the function f at x_0:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h},$$

which gives a way to generate an approximation to $f'(x_0)$ by computing:

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

for small values of h.

From Lagrange polynomial: To approximate $f'(x_0)$, suppose first that $x_0 \in (a, b)$, where $f \in C^2[a, b]$, and that $x_1 = x_0 + h$ for some h that is sufficiently small to ensure that $x_1 \in [a, b]$. We construct the first Lagrange polynomial $P(x)$ for f determined by x_0 and x_1, with its error term:

$$\begin{aligned} f(x) &= P(x) + \frac{(x - x_0)(x - x_1)}{2!} f''(\xi) \\ &= \frac{f(x_0)(x - x_0 - h)}{-h} + \frac{f(x_0 + h)(x - x_0)}{h} + \frac{(x - x_0)(x - x_0 - h)}{2} f''(\xi). \end{aligned}$$

For some $\xi(x)$ between x_0 and x_1. Differentiating the above equation gives:

$$\begin{aligned} f'(x) =& \frac{f(x_0 + h) - f(x_0)}{h} + D_x \left[\frac{(x - x_0)(x - x_0 - h)}{2} f''(\xi) \right] \\ =& \frac{f(x_0 + h) - f(x_0)}{h} + \frac{2(x - x_0) - h}{2} f''(\xi) + \frac{(x - x_0)(x - x_0 - h)}{2} D_x \left(f''(\xi) \right). \end{aligned}$$

Deleting the terms involving ξ gives (As they are smaller by an order of h, if you set $x = x_0$ *or* x_1):

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

The terms involving ξ are higher-order remainder terms.

Remark: One problem with this formula is that we have no information about $D_x f''(\xi)$, so the truncation error cannot be estimated explicitly. When x is x_0, however, the coefficient of $D_x f''(\xi(x))$ is 0 , and the formula simplifies to:

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

For small values of h $(x = x_1)$, the difference quotient $[f(x_0 + h) - f(x_0)]/h$ can be used to approximate $f'(x_0)$ with an error bounded by $M|h|/2$, where M is a bound on $|f''(x)|$ for x between x_0 and $x_0 + h$. This formula is known as the forward-difference formula if $h > 0$ and the backward-difference formula if $h < 0$.

From Taylor expansion: Another way to derive this formula is via Taylor expansion: Let $f \in C^{p+1}$ near x, for any small h, we have:

$$f(x \pm h) = f(x) \pm hf'(x) + \frac{h^2}{2}f''(x) \pm \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) \pm \frac{h^5}{120}f^{(5)}(x) + \cdots .$$

Approximate $f'(x)$:

By combining different Taylor series expansions, we can obtain approximations of $f'(x)$ of various orders. For instance, subtracting the two expansions below:

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + h^2\tfrac{f''(x)}{2!} + h^3\tfrac{f'''(\xi_1)}{3!}, \quad \xi_1 \in (x, x+h) \\ f(x-h) &= f(x) - hf'(x) + h^2\tfrac{f''(x)}{2!} - h^3\tfrac{f'''(\xi_2)}{3!}, \quad \xi_2 \in (x-h, x) \end{aligned}$$

gives:

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

Then divide $2h$ on both sides, we have:

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

Here, $\frac{f(x+h)-f(x-h)}{2h}$ is an approximation of $f'(x)$ whose error is proportional to h^2. It is called the second-order or $O(h^2)$ centered difference approximation of $f'(x)$.

If we use expansions with more terms, higher-order approximations can be derived.

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + h^2\frac{f''(x)}{2!} + h^3\frac{f'''(x)}{3!} + h^4\frac{f^{(4)}(x)}{4!} + h^5\frac{f^{(5)}(\xi_1)}{5!}, \\ f(x-h) &= f(x) - hf'(x) + h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} + h^4\frac{f^{(4)}(x)}{4!} - h^5\frac{f^{(5)}(\xi_2)}{5!}, \\ f(x+2h) &= f(x) + 2hf'(x) + 4h^2\frac{f''(x)}{2!} + 8h^3\frac{f'''(x)}{3!} + 16h^4\frac{f^{(4)}(x)}{4!} + 32h^5\frac{f^{(5)}(\xi_3)}{5!}, \\ f(x-2h) &= f(x) - 2hf'(x) + 4h^2\frac{f''(x)}{2!} - 8h^3\frac{f'''(x)}{3!} + 16h^4\frac{f^{(4)}(x)}{4!} - 32h^5\frac{f^{(5)}(\xi_4)}{5!}. \end{aligned}$$

To eliminate the h^2 and h^3 terms, we do

$$8\big[f(x+h) - f(x-h)\big] - \big[f(x+2h) - f(x-2h)\big],$$

After rearranging terms results a $4th$ order centered difference approximation of $f'(x)$.

Method of undetermined coefficients

To derive a finite difference approximation for arbtrary order, we use the method of undetermined coefficients. Suppose we want to approximate $f'(x)$ by a linear combination of function values at $x + kh$ for some stencil points $k = k_1, k_2, \ldots, k_m$:

$$f'(x) \approx c_1 f(x + k_1 h) + c_2 f(x + k_2 h) + \cdots + c_m f(x + k_m h).$$

Our goal is to find the coefficients c_i such that the approximation is as accurate as possible. We follow the steps below:

1. Taylor expand each stencil point.

For each i, expand $f(x+k_ih)$ about x using a Taylor series:

$$f(x+k_ih) = f(x) + k_ihf'(x) + \frac{(k_ih)^2}{2!}f''(x) + \frac{(k_ih)^3}{3!}f^{(3)}(x) + \cdots .$$

2. Substitute expansions into the linear combination.

$$\begin{aligned} c_1f(x+k_1h) + \cdots + c_mf(x+k_mh) = &\left(\sum_{i=1}^{m} c_i\right) f(x) + \left(\sum_{i=1}^{m} c_ik_i\right) hf'(x) \\ &+ \left(\sum_{i=1}^{m} c_ik_i^2\right) \frac{h^2}{2!}f''(x) + \left(\sum_{i=1}^{m} c_ik_i^3\right) \frac{h^3}{3!}f^{(3)}(x) + \cdots . \end{aligned}$$

3. Match coefficients for accuracy.

We require that this expression equals $f'(x) + \mathcal{O}(h^p)$ for some order p. This leads to a system of equations for c_i by matching coefficients of $f(x), f''(x), f^{(3)}(x), \ldots$ to eliminate undesired terms:

$$\begin{cases} \displaystyle\sum_{i=1}^{m} c_i = 0 & \text{(eliminate constant term)}, \\ \displaystyle\sum_{i=1}^{m} c_ik_i = \frac{1}{h} & \text{(match } f'(x) \text{ term)}, \\ \displaystyle\sum_{i=1}^{m} c_ik_i^2 = 0 & \text{(eliminate } f''(x) \text{ term for higher order)}, \\ \displaystyle\sum_{i=1}^{m} c_ik_i^3 = 0, \quad \ldots & \end{cases}$$

4. Solving this linear system for the coefficients c_i.

Example 1: 3 point centered difference scheme for $f'(x)$.

Choose $k = \{-1, 0, +1\}$ with unknown coefficients c_{-1}, c_0, c_{+1}:

$$f'(x) \approx c_{-1}f(x-h) + c_0f(x) + c_{+1}f(x+h).$$

The system of equations becomes

$$\begin{cases} c_{-1} + c_0 + c_{+1} = 0, \\ -c_{-1} + 0 \cdot c_0 + c_{+1} = \frac{1}{h}, \\ c_{-1} + 0 \cdot c_0 + c_{+1} = 0. \end{cases}$$

Solving gives $c_{-1} = -\dfrac{1}{2h}$, $c_0 = 0$, $c_{+1} = \dfrac{1}{2h}$, i.e.,

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$

which is the familiar second-order centered difference scheme.

What if I choose k=-1,0,+1,+2? What is the order of accuracy?

For the stencil $k = \{-1, 0, 1, 2\}$, the unknowns are $c_{-1}, c_0, c_{+1}, c_{+2}$), which can be obtained by matching coefficients.

$$\begin{cases} c_{-1} + c_0 + c_{+1} + c_{+2} = 0, \\ (-1)c_{-1} + 0 \cdot c_0 + 1 \cdot c_{+1} + 2c_{+2} = \frac{1}{h}, \\ (-1)^2 c_{-1} + 0^2 c_0 + 1^2 c_{+1} + 2^2 c_{+2} = 0, \\ (-1)^3 c_{-1} + 0^3 c_0 + 1^3 c_{+1} + 2^3 c_{+2} = 0. \end{cases}$$

Solving it, yields:

$$c_{-1} = -\frac{1}{3h}, \quad c_0 = -\frac{1}{2h}, \quad c_{+1} = \frac{1}{h}, \quad c_{+2} = -\frac{1}{6h}.$$

So the finite difference formula of $O(h^3)$ is:

$$f'(x) \approx \frac{-2f(x-h) - 3f(x) + 6f(x+h) - f(x+2h)}{6h}.$$

Now we can use these derived schemes to approximate $f'(x)$, given some data points. There are two examples on Page 179 on Burden's book.

3.1.2 Finite difference schemes and its analysis

For 1st order derivatives:

1> Forward difference of $1st$ order O (h):

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{h}{2} f''(x) + \frac{h^2}{6} f^{(3)}(x) + O\left(h^3\right).$$

Some simple analysis on error and step size:

Numerically, evaluating $f(x+h) - f(x)$ suffers from cancellation when h is very small. Let $\tilde{f}$ be the nearby representable floating–point number, we have:

$$\tilde{f}(x) = f(x)\left(1 + \delta_1\right), \quad \tilde{f}(x+h) = f(x+h)\left(1 + \delta_2\right), \quad |\delta_i| \lesssim \varepsilon.$$

The actual subtraction is:

$$\tilde{f}(x+h) - \tilde{f}(x) = \underbrace{[f(x+h) - f(x)]}_{\text{true difference} \approx h f'(x)} + \underbrace{f(x+h)\delta_2 - f(x)\delta_1}_{\text{round-off error } \approx O(|f(x)|)(\delta_2 - \delta_1) = O(|f(x)|)\epsilon}.$$

Therefore, the floating-point subtraction introduces a relative round-off error of order machine epsilon ε. After division by h, the round-off contribution behaves like:

$$\text{Round-off error } = O\left(\frac{\varepsilon |f(x)|}{h}\right) = \frac{C_1}{h}.$$

From the Taylor expansion, the truncation error is:

$$\text{Truncation error } = \frac{h}{2} f''(x) = C_2 h.$$

Therefore, the total error is:

$$\text{Error}(h) = \frac{C_1}{h} + C_2 h.$$

To minimize the error, we differentiate the above equation wrt to h, and then set it equal to 0, we obtain:

$$\frac{dError}{dh} = -\frac{C_1}{h^2} + C_2 = 0.$$

Namely, $h = \sqrt{C_1/C_2}$ should be the optimal h. Since $C_1 \sim \varepsilon|f(x)|$ and $C_2 = \frac{1}{2}|f''(x)|$, we have:

$$h \propto \sqrt{\varepsilon}.$$

which gives the best h in forward difference scheme.

Remark: If you make h too big $\rightarrow$ truncation dominates. If you make h too small $\rightarrow$ round-off dominates. The sweet spot is around $\sqrt{\varepsilon}$, which in double precision is about 10^{-8}.

2> Backward difference of O (h):

$$\frac{f(x) - f(x-h)}{h} = f'(x) - \frac{h}{2}f''(x) + \frac{h^2}{6}f^{(3)}(x) + O\left(h^3\right).$$

Similar to the error analysis for forward difference.

3> Central difference O (h^2):

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{6}f^{(3)}(x) + \frac{h^4}{120}f^{(5)}(x) + O\left(h^6\right).$$

Error analysis for central difference:

The truncation error scales like C_2h^2 with

$$C_2 = \frac{1}{6}\left|f^{(3)}(x)\right|,$$

and round-off error is the difference divided by $2h$ which behaves as $O\left(\frac{\varepsilon|f(x)|}{2h}\right) = \frac{C_1}{h}$. Therefore,

$$\text{Total error} = C_2h^2 + \frac{C_1}{h}.$$

Differentiate it and set it to 0, we have:

$$h^3 = C_1/C_2, \quad \Rightarrow h \propto \sqrt[3]{\varepsilon}.$$

4> Central difference with five points: O (h^4)

$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}.$$

Round-off error:

$$\frac{\varepsilon(|-1| + |8| + |-8| + |1|)|f(x)|}{12h} = \frac{18}{12h}\varepsilon|f(x)| = fracC_1h,$$

Truncation error:

$$Error = C_2h^4.$$

Therefore, the optimal step size $h \propto \sqrt[5]{\varepsilon}$.

For high order derivatives:

Central scheme of 3 points O (h^2):

$$\frac{f(x+h)-2f(x)+f(x-h)}{h^2}=f''(x)+\frac{h^2}{12}f^{(4)}(x)+O\left(h^4\right).$$

Central scheme of 5 points O (h^4) :

$$f''(x)\approx\frac{-f(x+2h)+16f(x+h)-30f(x)+16f(x-h)-f(x-2h)}{12h^2},$$

***Compact finite difference schemes:**(I am using a different notation here.)

Compact finite difference schemes are higher order discretizations that achieve spectral like resolution using implicit relations between neighboring points. For example, suppose you want the first derivative $u'(x_i)$. A six-order tri-diagonal compact scheme might look like:

$$\frac{1}{3}u'_{i-1}+u'_i+\frac{1}{3}u'_{i+1}=\frac{14}{9h}\left(u_{i+1}-u_{i-1}\right)+\frac{1}{9h}\left(u_{i+2}-u_{i-2}\right)$$

where h is the grid spacing.

To derive the formula, write out the taylor expansion of u'_{i-1} and u'_{i+1}, then match the coefficients that has the same order (using mathod of undetermined coefficients).

Example 1: 1D Poisson equation on [0, L]:

$$-u_{xx}=f,\quad u(0)=0(\text{Dirichlet}),\quad u'(L)=0(\text{Neumann}).$$

We choose an uniform grid $x_i=ih$ with $h=L/N$, and $i=0,\ldots,N$. The unknowns are $u_1,\ldots,u_N$ at those "N" nodes, given that $u_0=0$ is known from BC. We can discretize the ODE using the central difference scheme. For all the interior nodes $i=1,\ldots,N-1$, we have:

$$-\frac{u_{i-1}-2u_i+u_{i+1}}{h^2}=f_i\quad\Rightarrow\quad-u_{i-1}+2u_i-u_{i+1}=h^2f_i.$$

For the point at $x_N=L$, we can use one sided 2nd order difference scheme:

$$u'(L)\approx\frac{3u_N-4u_{N-1}+u_{N-2}}{2h}=0\Longleftrightarrow u_{N-2}-4u_{N-1}+3u_N=0.$$

For the point at $x_0=0$, we can either use the center difference with u_0 set to 0 to have:

$$2u_1-u_2=h^2f_1.$$

or directly set $u_0=0$.

The result linear system is:

$$\underbrace{\begin{bmatrix}2&-1&0&\cdots&0&0&0\\-1&2&-1&\cdots&0&0&0\\0&-1&2&\ddots&0&0&0\\\vdots&\ddots&\ddots&\ddots&2&-1&0\\0&0&...&0&-1&2&-1\\0&0&0&...&1&-4&3\end{bmatrix}}_{A}\underbrace{\begin{bmatrix}u_1\\u_2\\u_3\\\vdots\\u_{N-1}\\u_N\end{bmatrix}}_{\mathbf{u}}=\underbrace{\begin{bmatrix}h^2f_1\\h^2f_2\\h^2f_3\\\vdots\\h^2f_{N-1}\\0\end{bmatrix}}_{\mathbf{b}}$$

where A is tri-diagonal matrix, however this is not a semi-positive definite matrix due to the last line, the symmetry is lost. To remedy it, we can use the central differnce scheme for ths last row, namely:

$$-\frac{u_{N-1}-2u_N+u_{N+1}}{h^2}=f_N,\quad u_{N+1}=u_{N-1}\Rightarrow -2u_{N-1}+2u_N=h^2f_N,$$

with a ghost node u_{N+1}. The result system becomes:

$$\underbrace{\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 & 0 & 0\\ -1 & 2 & -1 & \cdots & 0 & 0 & 0\\ 0 & -1 & 2 & \ddots & 0 & 0 & 0\\ \vdots & \ddots & \ddots & \ddots & 2 & -1 & 0\\ 0 & 0 & ... & 0 & -1 & 2 & -1\\ 0 & 0 & 0 & ... & 0 & -1 & 1 \end{bmatrix}}_{A}\underbrace{\begin{bmatrix} u_1\\ u_2\\ u_3\\ \vdots\\ u_{N-1}\\ u_N \end{bmatrix}}_{\mathbf{u}}=\underbrace{\begin{bmatrix} h^2f_1\\ h^2f_2\\ h^2f_3\\ \vdots\\ h^2f_{N-1}\\ h^2f_N/2 \end{bmatrix}}_{\mathbf{b}}$$

which is SPD (Conjugate gradient method can be used, we will learn this in Chapter 4.)

A good practice: You can apply this approach to solve other ODEs, such as $u_{xx}+u_x=f$. For example, use the central difference scheme:

$$\frac{u_{i-1}-2u_i+u_{i+1}}{h^2}+\frac{u_{i+1}-u_{i-1}}{2h}=f_i.$$

The result system is also a tri-diagonal system (but not symmetric!):

$$A\mathbf{u}=\mathbf{b},\quad A=\begin{bmatrix} -2 & 1+\frac{h}{2} & 0 & \cdots & 0\\ 1-\frac{h}{2} & -2 & 1+\frac{h}{2} & \cdots & 0\\ 0 & 1-\frac{h}{2} & -2 & \ddots & 0\\ \vdots & \ddots & \ddots & \ddots & 1+\frac{h}{2}\\ 0 & \cdots & 0 & 2 & -2 \end{bmatrix},\quad \mathbf{b}=h^2\begin{bmatrix} f_1\\ f_2\\ \vdots\\ f_{N-1}\\ f_N \end{bmatrix}.$$

Example 2: 1D Heat Equation

Let's consider 1d heat equation:

$$u_t=\alpha u_{xx},\quad \alpha>0,$$

with grid spacing Δx and timestep Δt.

Forward in time (explicit), Central difference in space

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=\alpha\frac{u_{j+1}^n-2u_j^n+u_{j-1}^n}{\Delta x^2},$$
$$u_j^{n+1}=u_j^n+r\left(u_{j+1}^n-2u_j^n+u_{j-1}^n\right).$$

Define

$$r=\alpha\frac{\Delta t}{\Delta x^2}.$$

Accuracy: $O(\Delta t)+O(\Delta x^2)$.
Stability: $\alpha\frac{\Delta t}{\Delta x^2}\le\frac{1}{2}$.

To derive the above stability condition, we will use the von Neumann stability analysis (Its idea is based on Fourier decomposition of numerical error.) Thinking about how error relates to stability of a scheme, the error propagates in each time step do not significantly magnified but remain constant or decay, will results a stable scheme.

Assume the error can be represented as a sum of Fourier series.

$$\epsilon(x,t) = \sum_k E_k(t) e^{ikx}$$

where, k is the wavenumber (spatial frequency), e^{ikx} is the spatial oscillation, and $E_k(t)$ is the amplitude of that mode, which change in time. And notice that the behavior of each term of the series is the same as series itself, because the PDE and the finite difference scheme are always linear with constant coefficients, each Fourier mode evolves independently. We can just analyze one mode at a time, and the error is:

$$\epsilon_k(x,t) = E_k(t) e^{ikx},$$

where $E_k(t)$ is the error from time.

From the Forward, Central differece scheme, we can write out the error as:

$$\epsilon_j^{n+1} = \epsilon_j^n + r\left(\epsilon_{j+1}^n - 2\epsilon_j^n + \epsilon_{j-1}^n\right),$$

where $\epsilon_j^n = u_j^n - u\left(x_j, t_n\right)$. By taking a single mode of ϵ_j^n, we get:

$$\epsilon_j^n = E_k^n e^{ikx_j} = E_k^n e^{ikj\Delta x}.$$

and the corresponding shifted neighbors are:

$$\epsilon_{j+1}^n = E_k^n e^{ik(j+1)\Delta x} = E_k^n e^{ikj\Delta x} e^{ik\Delta x},$$

$$\epsilon_{j-1}^n = E_k^n e^{ik(j-1)\Delta x} = E_k^n e^{ikj\Delta x} e^{-ik\Delta x}.$$

Substituting into error equation above, yields:

$$E_k^{n+1} e^{ikj\Delta x} = E_k^n e^{ikj\Delta x} + r E_k^n e^{ikj\Delta x}\left(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\right).$$

The term $e^{ikj\Delta x}$ can be canceled on both sides, and thus the corresponding error behavior for a single fourier error term is:

$$E_k^{n+1} = E_k^n\left[1 + r\left(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\right)\right].$$

In order to have a stable scheme, the term $|\left\{1 + r\left(e^{ik_m\Delta x} - 2 + e^{-ik_m\Delta x}\right)\right\}|$on the RHS is like an amplification factor, which should be less than 1 to result a stable scheme. By introducing $\theta = k_m\Delta x \in [-\pi, \pi]$ and using the identities:

$$\sin\left(\frac{\theta}{2}\right) = \frac{e^{i\theta/2} - e^{-i\theta/2}}{2i} \quad \rightarrow \quad \sin^2\left(\frac{\theta}{2}\right) = -\frac{e^{i\theta} + e^{-i\theta} - 2}{4}$$

we have:

$$\frac{E_m(t+\Delta t)}{E_m(t)} = 1 - 4r\sin^2\left(\frac{k_m\Delta x}{2}\right),$$

and we defined the amplification factor G, which depends on k_m:

$$G = 1 - 4r \sin^2\left(\frac{k_m \Delta x}{2}\right).$$

Since G is real, stability $|G| \leq 1$ requires the worst case (highest wavenumber, $k_m \Delta x = \pi$), which gives the greatest value of $sin^2\left(\frac{k_m \Delta x}{2}\right)$ (Namely, to ensure r small enough to satisfy the condition for the most restrictive case) to satisfy:

$$G_{\min} = 1 - 4r \geq -1 \Longrightarrow r \leq \frac{1}{2},$$

and also $r = \frac{\alpha \Delta t}{\Delta x^2} \geq 0$ (otherwise $G > 1$ for small θ). Hence,

$$0 \leq r \leq \frac{1}{2}.$$

```
clear; clc;

% Parameters
alpha = 0.1;
L = pi;
T = 5.0;

Nx = 500;
dx = L/Nx;
x = linspace(0,L,Nx+1).';

% stability requires r<0.5
r = 0.501; %Try r>0.5 unstable
dt = r*dx^2/alpha;

Nt = ceil(T/dt);
dt = T/Nt;
r = alpha*dt/dx^2;

u = sin(x); % initial condition u(x,0) = sin(x)
u(1) = 0; u(end) = 0; % BCs, both ends =0

% Time loop
u = sin(x); u([1 end]) = 0;

for n = 1:Nt
   u_new = u;
   u_new(2:end-1) = u(2:end-1) + r * (u(3:end) - 2*u(2:end-1) + u(1:end-2));
   u_new(1) = 0;
   u_new(end) = 0;
   u = u_new;
   if (mod(n,500)==1)
     plot(x, u, 'b-', 'LineWidth', 1.8); hold on;
   end
end
```

```
36
37 u_exact = exp(-alpha*T)*sin(x);
38
39 plot(x, u_exact, 'k--', 'LineWidth', 1.5); hold on;
40 xlabel('x'); ylabel('u(x,T)');
41 legend('exact: e^{-\alpha T}\sin x', 'FTCS');
42 title(sprintf('FTCS heat eqn: \\alpha=%.2f, T=%.1f, r=%.3f, Nx=%d', alpha, T, r,
       Nx));
43 grid on;
```

Backward in time (implicit), Central difference in space

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \alpha \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{\Delta x^2}.$$

This yields a tri-diagonal linear system.
Accuracy: $O(\Delta t) + O(\Delta x^2)$.
Stability: unconditional stable (means you can choose large time step).

From the BTCD scheme above, we can derive:

$$\begin{aligned} u_j^{n+1} - u_j^n &= r\left(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}\right), \\ u_j^{n+1} - ru_{j+1}^{n+1} + 2ru_j^{n+1} - ru_{j-1}^{n+1} &= u_j^n, \\ -ru_{j+1}^{n+1} + (1+2r)u_j^{n+1} - ru_{j-1}^{n+1} &= u_j^n. \end{aligned}$$

Let $\mathbf{u}^n = \left[u_1^n, \ldots, u_{M-1}^n\right]^\top$ be interior unknowns (assuming the total nodes are from $j = 0$ to M). Then at each time step, we solve the following system:

$$A\mathbf{u}^{n+1} = \mathbf{b}, \quad \mathbf{b} = \mathbf{u}^n$$

with the main diagonal entry of the system matrix A equals to $1 + 2r$, and sub-diagonal and super-diagonal equal to $-r$,

$$A = \begin{bmatrix} 1+2r & -r & & & \\ -r & 1+2r & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 1+2r & -r \\ & & & -r & 1+2r \end{bmatrix}.$$

For simplicity, let consider the Dirichlet BCs: $u_0^{n+1} = u_L, u_M^{n+1} = u_R$. The first and last rows above get constants added to the RHS $\mathbf{b}$, namely: For $j = 1$, we have:

$$(1+2r)u_1^{n+1} - ru_2^{n+1} = u_1^n + ru_L,$$

For row $j = M - 1$:

$$-ru_{M-2}^{n+1} + (1+2r)u_{M-1}^{n+1} = u_{M-1}^n + ru_R.$$

Notice that we still have a SPD matrix!

```
%BTCS example
clear; clc;

% Parameters
alpha = 0.1;
L = pi;
T = 5.0;

Nx = 500;
dx = L/Nx;
x = linspace(0, L, Nx+1).';

r = 1.5; % we can choose r freely (Because BTCS is unconditionally stable)
dt = r*dx^2/alpha;
disp("dt="+dt+"; dx="+dx);

Nt = ceil(T/dt);
dt = T/Nt;
r = alpha*dt/dx^2;

% IC u(x,0) = sin(x), BC u(0,t)=u(L,t)=0
u = sin(x); u([1 end]) = 0;

A = zeros(Nx+1, Nx+1);

for i = 1:Nx+1
   if i == 1
      % Left Dirichlet BC
      A(i,i) = 1;
   elseif i == Nx+1
      % Right
      A(i,i) = 1;
   else
      % Interior points
      A(i,i-1) = -r;
      A(i,i) = 1 + 2*r;
      A(i,i+1) = -r;
   end
end



% time loop
for n = 1:Nt
   b = u;
   b(1) = 0; % enforce BCs
   b(end) = 0;
   u = A \ b;
   if mod(n,500)==1
      plot(x, u, 'b-', 'LineWidth', 1.8); hold on;
```

```
   end
end

% Exact solution for comparison
u_exact = exp(-alpha*T)*sin(x);

plot(x, u_exact, 'k--', 'LineWidth', 1.5); hold on;
xlabel('x'); ylabel('u(x,T)');
legend('BTCS', 'exact: e^{-\alpha T}\sin x', 'Location', 'best');
title(sprintf('BTCS heat eqn: \\alpha=%.2f, T=%.1f, r=%.3f, Nx=%d', alpha, T, r,
    Nx));
grid on;
```

Crank-Nicolson scheme (implicit in time):

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \alpha \frac{(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (u_{j+1}^n - 2u_j^n + u_{j-1}^n)}{2\,\Delta x^2}.$$

Accuracy: $O(\Delta t^2) + O(\Delta x^2)$.
Stability: unconditional stable.

Provide the corresponding IC and BCs, you can solve for u_j^n. A good practice is to implement the scheme in Matlab, by following two previous examples.

Example 3: Linear Advection Equation

We consider the linear advection equation

$$u_t + c\,u_x = 0,$$

with initial condition

$$u(x, 0) = u_0(x).$$

The general solution is a rigid translation of the initial profile:

$$u(x, t) = u_0(x - ct),$$

namely, the wave profile u_0 propagates to the right with speed c if $c > 0$, and to the left with speed $|c|$ if $c < 0$.

Proof: The characteristic curves satisfy:

$$\frac{dx}{dt} = c \quad \Longrightarrow \quad x - ct = \text{constant},$$

and along such a curve, we have:

$$\frac{d}{dt}u(x(t), t) = u_t + \frac{dx}{dt}u_x = u_t + cu_x = 0,$$

which shows that u is constant (doesn't change in time) along characteristics. Therefore, we have:

$$u(x(t), t) = u(x_0, 0) = u_0(x_0),$$

so the solution of $u(x(t), t)$ depends only on the initial condition at the foot of the characteristic, $x_0 = x - ct$.

Intuitive choice FTCD

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -\,c\,\frac{u_{j+1}^n - u_{j-1}^n}{2\,\Delta x}. \tag{1}$$

rerange to get:

$$u_j^{n+1} = u_j^n - \frac{\nu}{2}\Big(u_{j+1}^n - u_{j-1}^n\Big), \tag{2}$$

where ν is called Courant number:

$$\nu = \frac{c\,\Delta t}{\Delta x}.$$

This is an unstable scheme!!

Upwind scheme (for $c > 0$, from left to right)

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -c\frac{(u_j^n - u_{j-1}^n)}{\Delta x}.$$

rerange the terms, we get:

$$u_j^{n+1} = u_j^n - \nu\,(u_j^n - u_{j-1}^n).$$

where ν is called Courant number:

$$\nu = \frac{c\,\Delta t}{\Delta x}$$

Accuracy: $O(\Delta t) + O(\Delta x)$.
Stability: $\nu \leq 1$.

```
%1D advection upwind
%Upwind scheme for linear advection
clear; clc;

c = -1.0; % advection speed
L = 2*pi; % domain length [0,L)
N = 200;
CFL = 0.9; % <=1 for stability

dx = L / N;
x = (0:N-1) * dx;

% IC
u = sin(x);

% Final time, and after T, exact solution returns to sin(x)
T = 5*2*pi / abs(c);

dt = CFL * dx / abs(c);
numSteps = ceil(T/dt);
dt = T/numSteps;
nu = c*dt/dx; % Courant number (positive here)

% plotting setup
```

```
figure;
plt = plot(x,u,'-','LineWidth',1.5); hold on;
ex = plot(x,sin(x),'--'); hold off;
legend('numerical','exact (init)');
xlabel('x'); ylabel('u'); title('Upwind advection (c>0)');
drawnow;

% Time loop
for n = 1:numSteps
   uL = circshift(u,+1); % u_{j-1} with periodic BC
   u = u - nu * (u - uL);

   if mod(n, max(1,round(numSteps/50)))==0 || n==numSteps
      set(plt,'YData',u);
      set(ex, 'YData', sin(x - c*(n*dt))); % exact solution
      title(sprintf('t = %.3f', n*dt));
      drawnow;
   end
end

% error analysis, try refine the dt and see how error changes
u_exact = sin(x - c*T);
err_inf = max(abs(u - u_exact));
err_L2 = sqrt(mean((u - u_exact).^2));
fprintf('CFL=%.3f, steps=%d, L_inf=%.3e, L2 norm=%.3e\n', ...
      nu, numSteps, err_inf, err_L2);
```

Lax-Friedrichs scheme

We start from the forward difference in time and center difference in space:

$$\begin{aligned} u_j^{n+1} &= u_j^n - \frac{\nu}{2}\left(u_{j+1}^n - u_{j-1}^n\right), \\ &= \frac{u_{j+1}^n + u_{j-1}^n}{2} - \frac{\nu}{2}\left(u_{j+1}^n - u_{j-1}^n\right). \end{aligned}$$

Therefore,

$$u_j^{n+1} = \tfrac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \tfrac{\nu}{2}\left(u_{j+1}^n - u_{j-1}^n\right).$$

Accuracy: $O(\Delta t) + O(\Delta x)$.

Stability: $|\nu| \leq 1$ (Compare to Upwind, no upwind switch is needed), but the scheme is diffusive (due to the artificial viscosity introduced).

```
% Lax-Friedrichs
% only need to modify the lines in te Time loop

 uR = circshift(u,-1); % u_{j+1}
 uL = circshift(u,+1); % u_{j-1}
 u = 0.5*(uR + uL) - 0.5*nu*(uR - uL) ;
```

Comparison of Upwind and Lax-Friedrichs

1> (Robustness) One symmetric stencil, doesn't require to determine the sign of c. In special scenario, if c(x,t) changes sign in space and time, Lax-Friedrichs will still work, where Upwind schemes fails.

2> (Artificial diffusion) Lax-Friedrichs smooths out the sharp features (Upwind provides sharper solutions).

Now, the new question is that how can we achieve higher order stable scheme?

Lax-Wendroff scheme

$$u_j^{n+1} = u_j^n - \frac{\nu}{2}(u_{j+1}^n - u_{j-1}^n) + \frac{\nu^2}{2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n).$$

Accuracy: Second order accurate in both space and time $\mathcal{O}(\Delta x^2 + \Delta t^2)$.
Stability: Stable if $|\nu| \leq 1$.

How do we derive it?

1> Taylor expand in time at (x_j, t^n) :

$$u\left(x_j, t^{n+1}\right) = u_j^n + \Delta t u_t^n + \frac{\Delta t^2}{2} u_{tt}^n + \mathcal{O}\left(\Delta t^3\right).$$

2> Replace time derivatives using the original PDE ($u_t = -cu_x$):

$$u_{tt} = \partial_t\left(-cu_x\right) = -cu_{xt} = -c\partial_x\left(u_t\right) = -c\partial_x\left(-cu_x\right) = c^2 u_{xx},$$

Therefore,

$$u\left(x_j, t^{n+1}\right) = u - c\Delta t u_x + \frac{c^2\Delta t^2}{2} u_{xx} + \mathcal{O}\left(\Delta t^3\right).$$

3> Discretize the spatial derivatives with centered differences:

$$u_x\left(x_j, t^n\right) \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}, \quad u_{xx}\left(x_j, t^n\right) \approx \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}.$$

Therefore, we have:

$$u_j^{n+1} = u_j^n - \frac{c\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{c^2\Delta t^2}{2\Delta x^2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right).$$

Let $\nu = \frac{c\Delta t}{\Delta x}$, then we have:

$$u_j^{n+1} = u_j^n - \frac{\nu}{2}\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{\nu^2}{2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right).$$

Example 3: Poisson Equation in 2D

We consider the 2d Poisson equation:

$$-(u_{xx} + u_{yy}) = f(x, y).$$

Using the 5 Point Stencil (central differences in 2D), we have:

$$-\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} - \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = f_{i,j}.$$

If we choose $\Delta x = \Delta y = h$, the formula can be simplified to:

$$-\frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} = f_{i,j}.$$

This gives a sparse linear system with an positive semi definite (PSD) matrix (all of its eigenvalues are nonnegative). This means that the scheme is stabile and physical consistent. In physics, it means that the discrete system never produces negative energy. For example, the temperature distribution distributes smoothly without oscillations and the total thermal energy can only dissipate or remain constant.

To construct the linear system, we will range the unknowns column wise:

$$\mathbf{u} = \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ \vdots \\ u_{m,1} \\ u_{1,2} \\ \vdots \\ u_{m,2} \\ \vdots \\ u_{1,m} \\ \vdots \\ u_{m,m} \end{bmatrix} \in \mathbb{R}^{m^2}.$$

The terms $-u_{i+1,j} - u_{i-1,j} + 2u_{i,j}$ produces a tri-diagonal block T_m of size $m \times m$:

$$T_m = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

The terms $-u_{i,j+1} - u_{i,j-1}$ couple each row to the row above and below. That means each block row of A couples to its neighbors via $-I_m$ blocks, where I_m is the identity matrix of size $m \times m$.

The overall system matrix A is a block tri-diagonal matrix:

$$A = \frac{1}{h^2} \begin{bmatrix} T_m & -I_m & 0 & \cdots & 0 \\ -I_m & T_m & -I_m & \cdots & 0 \\ 0 & -I_m & T_m & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -I_m \\ 0 & \cdots & 0 & -I_m & T_m \end{bmatrix}.$$

```
%Poisson equation in 2D, central difference scheme

clear; clc;

m = 100;
```

```
h = 1/(m+1); % dx
x = (1:m)*h; y = (1:m)*h; % interior nodes
[X,Y] = meshgrid(x,y);

u_exact = sin(pi*X).*sin(pi*Y);
f = 2*pi^2*u_exact; % because -u = f, with u=sin(pi x)sin(pi y)

i = ones(m,1);
T = spdiags([-i 2*i -i], -1:1, m, m); %matlab func to generate sparse matrix.
    You can type "help spdiags" in command window to see the usage
I = speye(m); % sparse identity matrix of dimension mxm
A = (kron(I,T) + kron(T,I)) / h^2;

b = f(:); % Dirichlet BC =0

u = A \ b;
U = reshape(u, m, m);

err = norm(U - u_exact, 'fro')*h; % L2 error

fprintf('m=%d, h=%.3g, L2 error=%.3e\n', m, h, err);
surf(X,Y,U), shading interp, title('Numerical u(x,y)')
xlabel x; ylabel y; zlabel u
```

An alternative approach is to use an iterative approach by introducing an artificial time:

$$U_{i,j}^{k+1} = \frac{1}{4}\left(U_{i+1,j}^{k} + U_{i-1,j}^{k} + U_{i,j+1}^{k} + U_{i,j-1}^{k} - h^2 F_{i,j}\right),$$

which stops when the norm of the following defined residual:

$$R_{i,j}^{k+1} = \frac{4U_{i,j} - U_{i+1,j} - U_{i-1,j} - U_{i,j+1} - U_{i,j-1}}{h^2} - F_{i,j},$$

is less than some tolerance.

```
%Poisson 2D, iterative approach, no system assembly is required

clear; clc;

m = 128; L = 1; h = L/(m+1);
x = linspace(0,L,m+2); y = linspace(0,L,m+2);
[X,Y] = meshgrid(x,y);

U = zeros(m+2,m+2); % unknown with zeros on boundary
F = -2*pi^2*sin(pi*X).*sin(pi*Y); % RHS for the test problem

tol = 1e-8; maxit = 5e4; res = Inf; it = 0;

while res > tol && it < maxit
   it = it + 1;

   % neighbor sum S = u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j-1)
   S = zeros(m+2,m+2);
```

```
    S(2:end-1,2:end-1) = U(3:end,2:end-1) + U(1:end-2,2:end-1) + ...
                    U(2:end-1,3:end) + U(2:end-1,1:end-2);

    Unew = U;
    Unew(2:end-1,2:end-1) = (S(2:end-1,2:end-1) - h^2*F(2:end-1,2:end-1))/4;

    % residual: (4U - S)/h^2 - F (since (4u - S)/h^2 = F at solution)
    Snew = zeros(m+2,m+2);
    Snew(2:end-1,2:end-1) = Unew(3:end,2:end-1) + Unew(1:end-2,2:end-1) + ...
                        Unew(2:end-1,3:end) + Unew(2:end-1,1:end-2);
    R = (4*Unew(2:end-1,2:end-1) - Snew(2:end-1,2:end-1))/h^2 - F(2:end-1,2:end
        -1);
    res = norm(R,'fro')*h;

    U = Unew;
end

Uex = sin(pi*X).*sin(pi*Y);
err = h*norm(U(2:end-1,2:end-1) - Uex(2:end-1,2:end-1), 'fro');

fprintf('Jacobi: it=%d, residual=%.3e, L2 err=%.3e\n', it, res, err);

figure; surf(X,Y,U); shading interp; title('Numerical simulation: u'); xlabel x;
     ylabel y;
figure; surf(X,Y,abs(U-Uex)); shading interp; title(sprintf('Abs error u-u_{
    exact}, L2=%.3e',err));
```

3.2 Richardson extrapolation

The main idea: Suppose a numerical approximation $N(h)$ to a quantity M (a derivative, an integral, etc.) has an asymptotic error expansion:

$$M = N(h) + K_1 h + K_2 h^2 + K_3 h^3 + \cdots,$$

for unknown constants K_j. If we also evaluate the same formula at a halved stepsize, $N(h/2)$, we can linearly combine $N(h)$ and $N(h/2)$ to cancel the leading error term and obtain a higher-order approximation to M:

Assume the asymptotic expansion:

$$M = N(h) + K_1 h + K_2 h^2 + K_3 h^3 + \cdots,$$

and also evaluate the same formula at half step size:

$$M = N\left(\frac{h}{2}\right) + K_1 \frac{h}{2} + K_2 \frac{h^2}{4} + K_3 \frac{h^3}{8} + \cdots.$$

Form the linear combination $2N(h/2) - N(h)$, we have:

$$2N\left(\frac{h}{2}\right) - N(h) = M + \frac{1}{2} K_2 h^2 + \frac{3}{4} K_3 h^3 + \cdots,$$

Notice that the $K_1 h$ term cancels. Hence, we have the Richardson extrapolation of M with order of $\mathcal{O}(h^2)$:

$$N_{\text{Rich}}(h) = 2N\left(\frac{h}{2}\right) - N(h).$$

Extension to general refinement ratio and leading order:

If the leading error scales as h^p :

$$M = N(h) + C_1 h^p + C_2 h^{p+1} + \cdots,$$

and you refine by a factor $r > 1$, then

$$M = N\left(\frac{h}{r}\right) + C_1\left(\frac{h}{r}\right)^p + C_2\left(\frac{h}{r}\right)^{p+1} + \cdots.$$

Now you have two equations for the same M, one at spacing h, one at h/r. Next, we can eliminate the $C_1 h^p$ term by doing a linear combination.

Multiply the refined approximation by a factor of r^p :

$$r^p M = r^p N\left(\frac{h}{r}\right) + C_1 h^p + C_2 \frac{h^{p+1}}{r} + \cdots.$$

Subtract the original expansion for M, we have:

$$r^p M - M = \left(r^p N\left(\frac{h}{r}\right) - N(h)\right) + \left(C_2\frac{h^{p+1}}{r} - C_2 h^{p+1}\right) + \cdots,$$

in which the $C_1 h^p$ terms cancel exactly. This gives the general Richardson extrapolation formula:

$$N_{\text{Rich}}(h) = \frac{r^p N\left(\frac{h}{r}\right) - N(h)}{r^p - 1},$$

whose error is $O\left(h^{p+1}\right)$ (Improved the order by 1).

Multi-level Richardson extrapolation on a refinement sequence $(h, h/r, h/r^2, \ldots)$: We can define the following notations of approximations:

$$R_{0,0} = N(h) = M - \left(C_1 h^p + C_2 h^{p+1} + C_3 h^{p+2} + \cdots\right),$$
$$R_{0,1} = N(h/r), \quad R_{0,2} = N\left(h/r^2\right), \ldots$$

and the recurrence formula for $R_{k,j}$:

$$R_{k,j} = \frac{r^{p+k-1} R_{k-1,j} - R_{k-1,j-1}}{r^{p+k-1} - 1}, \quad k \geq 1.$$

We can construct a lower-triangular table to store each level of $R_{k,j}$:

$$\begin{array}{ccccc} R_{0,0} & R_{0,1} & R_{0,2} & R_{0,3} & \cdots \\ & R_{1,1} & R_{1,2} & R_{1,3} & \cdots \\ & & R_{2,2} & R_{2,3} & \cdots \\ & & & R_{3,3} & \cdots \end{array},$$

where $R_{0,j}$ are the raw numerical approximations. $R_{1,j}$ are first-level extrapolations, and $R_{2,j}$ are second-level, and so on. For the 1st row (when $k = 0$), we calculate the direct numerical results:

$$R_{0,j} = N\left(\frac{h}{r^j}\right), \quad j = 0, 1, 2, \dots$$

Then, for higher rows, we use the formula:

$$R_{k,j} = \frac{r^{p+k-1}R_{k-1,j} - R_{k-1,j-1}}{r^{p+k-1} - 1}.$$

To list a few:

(1) When $k = 1$, we have:

$$R_{1,1} = \frac{r^p R_{0,1} - R_{0,0}}{r^p - 1}, \quad R_{1,2} = \frac{r^p R_{0,2} - R_{0,1}}{r^p - 1}.$$

Both are of $\mathcal{O}(h^{p+1})$.

Proof for $R_{1,1}$:

$$\begin{aligned} r^p R_{0,1} - R_{0,0} &= r^p\left(M - C_1 h^p r^{-p} - C_2 h^{p+1} r^{-(p+1)} - \cdots\right) - \left(M - C_1 h^p - C_2 h^{p+1} - \cdots\right) \\ &= \left(r^p - 1\right) M + 0 \cdot C_1 h^p + C_2 h^{p+1}\left(1 - r^{-1}\right) + O\left(h^{p+2}\right), \end{aligned}$$

thus

$$R_{1,1} = M + \frac{C_2 h^{p+1}\left(1 - r^{-1}\right)}{r^p - 1} + O\left(h^{p+2}\right) \quad \Rightarrow \quad R_{1,1} - M = O\left(h^{p+1}\right).$$

(2) When $k = 2$, we have:

$$R_{2,2} = \frac{r^{p+1}R_{1,2} - R_{1,1}}{r^{p+1} - 1}$$

which is of $\mathcal{O}(h^{p+2})$.

Proof:

$$R_{1,2} = M + C_1(\frac{h}{r})^{p+1} + C_2(\frac{h}{r})^{p+2} + \cdots, \quad R_{1,1} = M + C_1 h^{p+1} + C_2 h^{p+2} + \cdots$$

Then

$$r^{p+1}R_{1,2} - R_{1,1} = \left(r^{p+1} - 1\right) M + C_1\left(h^{p+1} - h^{p+1}\right) + C_2 h^{p+2}\left(r^{-1} - 1\right) + \cdots,$$

therefore,

$$R_{2,2} = M + \frac{B h^{p+2}\left(r^{-1} - 1\right)}{r^{p+1} - 1} + O\left(h^{p+3}\right) \quad \Rightarrow \quad R_{2,2} - M = O\left(h^{p+2}\right).$$

Each level increases an order of 1.

Example 1: Using the centered difference

$$N_0(h) = \frac{f\left(x_0 + h\right) - f\left(x_0 - h\right)}{2h}$$

to approximate $f'\left(x_0\right)$ for $f(x) = xe^x$ at $x_0 = 2$, given that:

$$N_0(0.2) = 22.414160, \quad N_0(0.1) = 22.228786, \quad N_0(0.05) = 22.182564.$$

Since the step sizes are: $h = 0.2, h/2 = 0.1, h/4 = 0.05$, we know that $r = 2$. Besides, the centered difference scheme is a second order scheme, we have:

$$N_0(h) = f'(x) - \left(C_1 h^2 + C_2 h^4 + C_3 h^6 + \cdots\right).$$

Expand about x_0 :

$$f(x_0 + h) = f + hf' + \frac{h^2}{2}f'' + \frac{h^3}{6}f^{(3)} + \frac{h^4}{24}f^{(4)} + \frac{h^5}{120}f^{(5)} + \cdots$$
$$f(x_0 - h) = f - hf' + \frac{h^2}{2}f'' - \frac{h^3}{6}f^{(3)} + \frac{h^4}{24}f^{(4)} - \frac{h^5}{120}f^{(5)} + \cdots$$

Subtract and divide by $2h$:

$$\frac{f(x_0 + h) - f(x_0 - h)}{2h} = f'(x_0) + \underbrace{\frac{h^2}{6}f^{(3)}(x_0)}_{\text{1st error term}} + \underbrace{\frac{h^4}{120}f^{(5)}(x_0)}_{\text{next}} + \cdots.$$

This means that $p = 2$ for row $k = 0$. Therefore, we can apply the Richardson to calculate $N_1(h)$ and $N_1(h/2)$, with $r = 2, p = 2, k = 1$, $(r^{p+k-1} = 2^2 = 4)$:

$$N_1(0.2) = N_0(0.1) + \frac{N_0(0.1) - N_0(0.2)}{3} = 22.166995,$$

and

$$N_1(0.1) = N_0(0.05) + \frac{N_0(0.05) - N_0(0.1)}{3} = 22.167157,$$

To calculate $N_2(h)$, we have $r = 2, p = 4, k = 2$, $(r^{p+k-1} = 2^5 = 32)$. So we have:

$$N_2(0.2) = N_1(0.1) + \frac{N_1(0.1) - N_1(0.2)}{31} = 22.167168,$$

which is essentially the exact derivative to six decimal places. The sequence $N_1 \to N_2 \to N_3$ steadily removes h^2, then h^4 error terms.

So the Richardson extrapolation can be treated as a postprocessing technique that accelerates convergence by combining numerical solutions at multiple step sizes (or mesh sizes) to cancel leading-order error terms.

3.3 Newton-Cotes formulas for integration

The strategy is to approximate the integral of $f(x)$ over an interval $[a, b]$ by evaluating the integrand at **equally spaced** points. Namely, divide the interval by n subintervals, with the width of $h = \frac{b-a}{n}$. The Newton-Cotes formula is given as follows:

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

where w_i are weights depending only on n, a, and b, and can be precomputed.

Formula derivation:

It is much easier to work out the derivation on a canonical grid by introducing the following mapping:

$$x = a + ht, \quad t \in [0, n = \frac{b-a}{h}].$$

Since $dx = hdt$, we have:

$$\int_a^b f(x)dx = \int_0^n f(a+ht)hdt = h\int_0^n g(t)dt.$$

By defining the function $g(t)$, such that $g(t) = f(a+ht)$ on those canonical grid, we can derive:

$$\int_a^b f(x)dx = h\int_0^n g(t)dt \approx \sum_{i=0}^{n} h\tilde{w}_i g(t_i),$$

Next, let p_n be the Lagrange interpolant of $g(t)$ at $t_0, \ldots, t_n$:

$$p_n(t) = \sum_{i=0}^{n} g\left(t_i\right)\ell_i(t), \quad \ell_i(t) = \prod_{\substack{m=0 \\ m\neq i}}^{n} \frac{t-m}{i-m}.$$

The Newton-Cotes rule integrates p_n exactly:

$$\int_a^b f(x)dx \approx h\int_0^n p_n(t)dt = h\sum_{i=0}^{n} g\left(t_i\right) \underbrace{\int_0^n \ell_i(t)dt}_{=:W_i}.$$

From the above equation, we can identify the weights:

$$w_i = hW_i, \quad W_i = \int_0^n \ell_i(t)dt = \int_0^n \prod_{\substack{m=0 \\ m\neq i}}^{n} \frac{t-m}{i-m}dt.$$

Example 1: Derive the Simpson's Rule for calculating $\int_a^b f(x)dx$ for $n = 2$.

Given $n = 2$, namely $t \in [0, 2]$. We have three nodes $(0, 1, 2)$, and the corresponding Lagrangians are:

$$\ell_0(t) = \frac{(t-1)(t-2)}{(0-1)(0-2)} = \frac{(t-1)(t-2)}{2},$$

$$\ell_1(t) = \frac{t(t-2)}{(1-0)(1-2)} = -t(t-2),$$

$$\ell_2(t) = \frac{t(t-1)}{(2-0)(2-1)} = \frac{t(t-1)}{2}.$$

Integrate each over $[0, 2]$, we get:

$$W_0 = \int_0^2 \ell_0(t)dt = \frac{1}{3}, \quad W_1 = \int_0^2 \ell_1(t)dt = \frac{4}{3}, \quad W_2 = \int_0^2 \ell_2(t)dt = \frac{1}{3}$$

Then we obtain the weights $w_i = hW_i$ with $h = (b-a)/2$. Therefore, the final integration formulas is:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

Below are some examples of the integration formulas.

(1). Trapezoidal Rule (n = 1):

$$\int_a^b f(x)dx \approx \frac{b-a}{2}[f(a)+f(b)].$$

(2). Simpson's Rule (n = 2):

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right].$$

(3). Simpson's 3/8 Rule ($n = 3$):

$$\int_a^b f(x)dx \approx \frac{3h}{8}\left[f\left(x_0\right)+3f\left(x_1\right)+3f\left(x_2\right)+f\left(x_3\right)\right].$$

Let $t \in [0,3]$ with nodes $0,1,2,3$. The Lagrange basis polynomials are:

$$\begin{aligned}
\ell_0(t) &= \frac{(t-1)(t-2)(t-3)}{(0-1)(0-2)(0-3)} = -\frac{(t-1)(t-2)(t-3)}{6},\\
\ell_1(t) &= \frac{t(t-2)(t-3)}{(1-0)(1-2)(1-3)} = \frac{t(t-2)(t-3)}{2},\\
\ell_2(t) &= \frac{t(t-1)(t-3)}{(2-0)(2-1)(2-3)} = -\frac{t(t-1)(t-3)}{2},\\
\ell_3(t) &= \frac{t(t-1)(t-2)}{(3-0)(3-1)(3-2)} = \frac{t(t-1)(t-2)}{6}.
\end{aligned}$$

Integrate each basis over $[0,3]$, we get W_i:

$$\begin{aligned}
W_0 &= \int_0^3 \ell_0(t)dt = \frac{3}{8}, \quad W_1 = \int_0^3 \ell_1(t)dt = \frac{9}{8},\\
W_2 &= \int_0^3 \ell_2(t)dt = \frac{9}{8}, \quad W_3 = \int_0^3 \ell_3(t)dt = \frac{3}{8}.
\end{aligned}$$

Map to $[a,b]$ with step $h = \frac{b-a}{3}$ and nodes $x_i = a + ht_i$, we have:

$$\int_a^b f(x)dx \approx h\left(\frac{3}{8}f\left(x_0\right)+\frac{9}{8}f\left(x_1\right)+\frac{9}{8}f\left(x_2\right)+\frac{3}{8}f\left(x_3\right)\right).$$

(4). Boole's Rule (n = 4):

$$\int_a^b f(x)dx \approx \frac{2h}{45}\left[7f\left(x_0\right)+32f\left(x_1\right)+12f\left(x_2\right)+32f\left(x_3\right)+7f\left(x_4\right)\right].$$

Remark: Notice that because the rule is built from polynomial interpolation, the Newton–Cotes formulas are **exact for polynomials up to degree n.** (Why? Thinking about the case when $f(x)$ is a polynomial of degree n or less, and the case when the degree is greater than n.)

Can we do better (achieving much higher accuracy) with the same number of function evaluation nodes? Answer is using "Gaussian quadrature".

3.4 Gaussian quadrature

For $\int_{-1}^{1} f(x)dx$, the n point Gauss-Legendre quadracture rule chooses nodes $\{x_i\}_{i=1}^n$ and weights $\{w_i\}_{i=1}^n$ so that:

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

is **exact for all polynomial of degree** $\leq 2n-1$ (maximal possible order with n points). How do we achieve that?

1> Nodes are chosen to be the zeros of a Legendre polynomial!

The central idea is to make use of orthogonal polynomials. Specifically, the Legendre polynomials $P_k(x)$ form a sequence of polynomials that are mutually orthogonal on $[-1, 1]$ with respect to the weight function $w(x) = 1$,

$$\int_{-1}^{1} P_k(x)P_m(x)w(x)dx = 0, \quad k \neq m.$$

Legendre polynomials $P_n(x)$ can also be defined as the solutions of Legendre equation:

$$\left(1 - x^2\right) y'' - 2xy' + n(n+1)y = 0.$$

So $y = P_n(x)$ satisfies

$$\left(1 - x^2\right) P_n''(x) - 2xP_n'(x) + n(n+1)P_n(x) = 0.$$

They can be obtained via recursion:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x). \tag{1}$$

To name a few here:

$$\begin{aligned}
P_0(x) &= 1, \\
P_1(x) &= x, \\
P_2(x) &= \frac{1}{2}\left(3x^2 - 1\right), \\
P_3(x) &= \frac{1}{2}\left(5x^3 - 3x\right), \\
P_4(x) &= \frac{1}{8}\left(35x^4 - 30x^2 + 3\right), \\
P_5(x) &= \frac{1}{8}\left(63x^5 - 70x^3 + 15x\right), \\
&\ldots
\end{aligned}$$

For the n point Gauss-Legendre rule, the quadrature nodes are chosen as the n distinct roots of the degree n Legendre polynomial $P_n(x)$:

$$P_n(x_i) = 0, \quad i = 1, \ldots, n, \quad x_i \in (-1, 1).$$

These roots are real, simple, and symmetric about the origin. Gauss–Legendre quadrature nodes never include the boundaries of the interval ($+1, -1$ excluded)!

Some useful identity:

The Legendre polynomials $P_n(x)$ satisfy:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x),$$

which holds for all $n \geq 1$. Solve for $xP_n(x)$, we have:

$$xP_n(x) = \frac{(n+1)P_{n+1}(x) + nP_{n-1}(x)}{2n+1}.$$

Differentiate the equation above wrt x, we obtain:

$$P_n(x) + xP'_n(x) = \frac{(n+1)P'_{n+1}(x) + nP'_{n-1}(x)}{2n+1}.$$

Next, solve for $P'_{n+1}(x)$:

$$P'_{n+1}(x) = \frac{(2n+1)\left[P_n(x) + xP'_n(x)\right] - nP'_{n-1}(x)}{n+1}.$$

Plug the expression of $P'_{n+1}(x)$ back into the differentiated expression, we have:

$$P_n(x)+xP'_n(x) = \frac{1}{2n+1}\left[(n+1)\left(\frac{(2n+1)\left[P_n(x) + xP'_n(x)\right] - nP'_{n-1}(x)}{n+1}\right) + nP'_{n-1}(x)\right].$$

After Simplification, we can derive a useful identity:

$$\left(1-x^2\right)P'_n(x) = n\left(P_{n-1}(x) - xP_n(x)\right).$$

2) Weights w_i

Using Lagrange basis at these roots and orthogonality of P_n, the weights are defined as follows:

$$w_i = \frac{2}{\left(1-x_i^2\right)\left[P'_n\left(x_i\right)\right]^2}, \quad i = 1, \ldots, n.$$

Derivation: Suppose we take the interpolating polynomial of $f(x)$ at the quadrature nodes $\{x_j\}$:

$$p(x) = \sum_{j=1}^{n} f\left(x_i\right)\ell_i(x),$$

where $\ell_i(x)$ are the Lagrange basis polynomials over Legendre zeros:

$$\ell_i(x) = \prod_{j\neq i}\frac{x-x_j}{x_i-x_j} = \frac{P_n(x)}{\left(x-x_i\right)P'_n\left(x_i\right)}.$$

Then, the quadrature can be approximated by calculating the integration of the Lagrangian interpolant:

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} p(x)dx = \int_{-1}^{1}\sum_{i=1}^{n} f\left(x_i\right)\ell_i(x)dx$$

Noticing that we can pull out the coefficients $f\left(x_i\right)$ from the integral, since it is a constant. We have:

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

To proof the expression of $w_i = \int_{-1}^{1} \ell_i(x)dx = \frac{2}{\left(1-x_i^2\right)[P_n'(x_i)]^2}, \quad i = 1, \ldots, n$, we can expand ℓ_i in the Legendre basis up to degree $n-1$ and then use orthogonality with P_{n-1}. Namely, given that $\deg \ell_i = n-1$, we can write:

$$\ell_i(x) = \alpha_i P_{n-1}(x) + \sum_{k=0}^{n-2} c_{ik} P_k(x).$$

Multiply both sides by $P_{n-1}(x)$ and integrate over $[-1, 1]$. Using the orthogonality property of Legendre polynomials, which satisfies that:

$$\int_{-1}^{1} P_m(x)P_k(x)dx = \begin{cases} 0, & m \neq k \\ \frac{2}{2m+1}, & m = k \end{cases}.$$

We can derive:

$$\int_{-1}^{1} \ell_i(x)P_{n-1}(x)dx = \alpha_i \int_{-1}^{1} P_{n-1}(x)^2 dx = \alpha_i \frac{2}{2(n-1)+1} = \alpha_i \frac{2}{2n-1}.$$

While we can also derive from the Guassian quadrature formula:

$$\int_{-1}^{1} \ell_i(x)P_{n-1}(x)dx = \sum_{j=1}^{n} w_j \ell_i(x_j) P_{n-1}(x_j) = w_i P_{n-1}(x_i).$$

Therefore, we have:

$$w_i P_{n-1}(x_i) = \alpha_i \frac{2}{2n-1}. \tag{2}$$

To obtain the coefficient α_i, since Legendre polynomial P_n has leading binomial coefficient of

$$\kappa_n = \frac{1}{2^n}\binom{2n}{n} = \frac{1}{2^n}\frac{(2n)!}{n!n!},$$

with leading term as:

$$P_n(x) = \kappa_n x^n + \cdots.$$

When divide $P_n(x)$ by $(x - x_i)$, the degree drops by one:

$$\frac{P_n(x)}{x - x_i} = \kappa_n x^{n-1} + (\text{ lower-degree terms })$$

So we know the expression of $\ell_i(x)$ is:

$$\ell_i(x) = \frac{P_n(x)}{(x - x_i)P_n'(x_i)} = \frac{\kappa_n x^{n-1}}{P_n'(x_i)} + \frac{(\text{ lower-degree terms })}{P_n'(x_i)},$$

with the coefficient of the leading term x^{n-1} in $\ell_i(x)$ is $\frac{\kappa_n}{P_n'(x_i)}$. Now, we can match the leading coefficients:

$$\alpha_i \kappa_{n-1} = \frac{\kappa_n}{P_n'(x_i)} \quad \Rightarrow \quad \alpha_i = \frac{\kappa_n}{\kappa_{n-1}} \frac{1}{P_n'(x_i)}.$$

Since:

$$\frac{\kappa_n}{\kappa_{n-1}} = \frac{2^{-n}\binom{2n}{n}}{2^{-(n-1)}\binom{2n-2}{n-1}} = \frac{2n-1}{n}.$$

We have:

$$\alpha_i = \frac{2n-1}{nP_n'(x_i)}.$$

Plug this α_i into (2):

$$w_i P_{n-1}(x_i) = \frac{2}{2n-1}\frac{2n-1}{nP_n'(x_i)} = \frac{2}{nP_n'(x_i)}.$$

And use the identity (derived already in the notes before):

$$\left(1-x^2\right)P_n'(x) = n\left(P_{n-1}(x) - xP_n(x)\right),$$

and evaluate at a root $x = x_i$ of P_n (since x_i is one of the zeros of $P_n(x_i) = 0$):

$$P_{n-1}(x_i) = \frac{1-x_i^2}{n}P_n'(x_i),$$

$$w_i\left(\frac{1-x_i^2}{n}P_n'(x_i)\right) = \frac{2}{nP_n'(x_i)} \quad \Longrightarrow \quad w_i = \frac{2}{\left(1-x_i^2\right)\left[P_n'(x_i)\right]^2}.$$

In the actual calculation scenario for the quadrature weights:

You have two numerically equivalent options; Approach 2 is often more convenient/stable in code:

1. Using the derivative:

$$w_i = \frac{2}{\left(1-x_i^2\right)\left[P_n'(x_i)\right]^2}$$

2. Avoiding the derivative (use the identity $\left(1-x^2\right)P_n'(x) = n\left(P_{n-1}(x) - xP_n(x)\right)$, and at a root x_i we have $P_n(x_i) = 0$):

$$P_n'(x_i) = \frac{n}{1-x_i^2}P_{n-1}(x_i) \Longrightarrow w_i = \frac{2\left(1-x_i^2\right)}{n^2\left[P_{n-1}(x_i)\right]^2}$$

This second form lets you compute w_i knowing only x_i and $P_{n-1}(x_i)$ (no derivative routine needed).

Why quadrature accuracy is of degree $2n-1$?

Let the nodes $x_1, \ldots, x_n$ be the roots of P_n on $(-1, 1)$, and the Lagrangian polynomial:

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

Define the following notations for quadrature and interpolant respectively:

$$Q_n[f] \;:=\; \sum_{i=1}^{n} w_i\, f(x_i), \qquad w_i \;=\; \int_{-1}^{1}\ell_i(x)\,dx,$$

and

$$I_n[f](x) \;=\; \sum_{i=1}^{n} f(x_i)\,\ell_i(x).$$

By linearity of the integral,

$$Q_n[f] \;=\; \sum_{i=1}^{n} w_i f(x_i) = \sum_{i=1}^{n} \left(\int_{-1}^{1} \ell_i(x)\,dx \right) f(x_i) = \int_{-1}^{1} \sum_{i=1}^{n} f(x_i)\ell_i(x)\,dx = \int_{-1}^{1} I_n[f](x)\,dx. \tag{1}$$

Let $f(x)$ be a polynomial with $\deg f \le 2n-1$. Divide by P_n:

$$f(x) \;=\; q(x)\,P_n(x) \;+\; r(x), \qquad \deg q \le n-1,\ \deg r \le n-1. \tag{2}$$

Regarding the quadrature of $f(x)$. Using linearity of $Q_n[f]$ and (2),

$$Q_n[f] = Q_n[qP_n] + Q_n[r] = \sum_{i=1}^{n} w_i\, q(x_i)P_n(x_i) \;+\; \sum_{i=1}^{n} w_i\, r(x_i).$$

Since $P_n(x_i) = 0$ for all nodes x_i, the first sum vanishes, we have:

$$\sum_{i=1}^{n} w_i\, q(x_i)P_n(x_i) = 0,$$

hence

$$Q_n[f] \;=\; \sum_{i=1}^{n} w_i\, r(x_i) \;=\; Q_n[r]. \tag{3}$$

Because $\deg r \le n-1$, $I_n[r] = r(x)$, so by (1), we have:

$$Q_n[r] \;=\; \int_{-1}^{1} I_n[r](x)\,dx = \int_{-1}^{1} r(x)\,dx,$$

and thus:

$$Q_n[f] \;=\; \int_{-1}^{1} r(x)\,dx. \tag{4}$$

While the true integral of $f(x)$ defined as integrating (2) over $[-1,1]$,

$$\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} q(x)P_n(x)\,dx + \int_{-1}^{1} r(x)\,dx. \tag{5}$$

Expand $q(x)$ in the Legendre basis up to degree $n-1$:

$$q(x) = \sum_{k=0}^{n-1} a_k\,P_k(x).$$

By orthogonality,

$$\int_{-1}^{1} q(x)P_n(x)\,dx = \sum_{k=0}^{n-1} a_k \int_{-1}^{1} P_k(x)P_n(x)\,dx = 0, \tag{6}$$

so

$$\int_{-1}^{1} f(x)\,dx = 0 + \int_{-1}^{1} r(x)\,dx = \int_{-1}^{1} r(x)\,dx. \tag{7}$$

Comparing (4) and (7), we have that:

$$Q_n[f] \;=\; \int_{-1}^{1} f(x)\,dx, \qquad \forall \text{ polynomials } f \text{ with } \deg f \le 2n-1.$$

3) Extend to the integration over a general interval $[a,b]$

Use $x = \frac{a+b}{2} + \frac{b-a}{2}t$ with $t \in [-1,1]$, we can map the interval from $[-1,1]$ to $[a,b]$. Then the general integral of $\int_a^b f(x)dx$ can be evaluated as follows:

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{a+b}{2} + \frac{b-a}{2}x_i\right).$$

The error of Gaussian quadrature (I will skip the proof..):

Let n be the number of nodes. For sufficiently smooth $f(x)$, we have:

$$E_n[f] := \int_{-1}^{1} f(x)dx - \sum_{i=1}^{n} w_i f(x_i) = (-1)^{n+1}\frac{2^{2n+1}(n!)^4}{(2n+1)((2n)!)^3} f^{(2n)}(\xi) \quad \text{for some } \xi \in (-1,1).$$

3.5 Romberg integration and Romberg Table

We want to approximate

$$I = \int_a^b f(x)dx.$$

we will perform the following steps:

1. Compute trapezoidal approximations $T(h), T(h/2), T(h/4), \ldots$.
2. Apply Richardson extrapolation:

$$R(k,0) = T\left(\frac{b-a}{2^k}\right), \quad k = 0,1,2,\ldots$$

$$R(k,j) = R(k,j-1) + \frac{1}{4^j-1}(R(k,j-1) - R(k-1,j-1)), \quad j = 1,2,\ldots,k$$

where the approximation improves as j increases. $R(k,k)$ is the Romberg estimate with very high accuracy.

Example 1: $\int_0^\pi \sin x\,dx$

The exact value of the integral is:

$$\int_0^\pi \sin x\,dx = 2.$$

1. Composite trapezoid values (first Romberg column)

Let T_n be the composite trapezoid rule with n equal panels on $[0,\pi]$.

1>. $R(0,0) = T_1$, with $h = \pi$:

$$T_1 = \frac{h}{2}\left[\sin 0 + \sin \pi\right] = 0.$$

2>. $R(1,0) = T_2$, with $h = \pi/2$, nodes $0, \pi/2, \pi$:

$$T_2 = \frac{\pi}{4}\left[0 + 2\sin(\pi/2) + 0\right] = \frac{\pi}{2} \approx 1.5707963268.$$

3>. $R(2,0) = T_4$, with $h = \pi/4$, nodes $0, \pi/4, \pi/2, 3\pi/4, \pi$:

$$\begin{aligned} T_4 &= \frac{\pi}{8}\Big[0 + 2\big(\sin(\tfrac{\pi}{4}) + \sin(\tfrac{\pi}{2}) + \sin(\tfrac{3\pi}{4})\big) + 0\Big] \\ &= \frac{\pi}{8}\left[2\left(\tfrac{\sqrt{2}}{2} + 1 + \tfrac{\sqrt{2}}{2}\right)\right] \\ &= \frac{\pi}{4}(1+\sqrt{2}) \approx 1.8961188979. \end{aligned}$$

2. Richardson extrapolation (Romberg updates) For $k \geq 1$ and $1 \leq j \leq k$,

$$R(k,j) = R(k,j-1) + \frac{R(k,j-1) - R(k-1,j-1)}{4^j - 1}.$$

We have:

$$R(1,1) = T_2 + \frac{T_2 - T_1}{3} = \frac{2\pi}{3} \approx 2.0943951024.$$

$$R(2,1) = T_4 + \frac{T_4 - T_2}{3} \approx 1.8961188979 + \frac{1.8961188979 - 1.5707963268}{3} \approx 2.0045597550.$$

$$R(2,2) = R(2,1) + \frac{R(2,1) - R(1,1)}{15} \approx 2.0045597550 + \frac{2.0045597550 - 2.0943951024}{15} \approx 1.9985707318.$$

3. Construct the Romberg table (up to $k = 2$)

	$j=0$	$j=1$	$j=2$
$k=0$	0		
$k=1$	1.5707963268	2.0943951024	
$k=2$	1.8961188979	2.0045597550	1.9985707318

As k and j increase, the values converge rapidly to the exact result 2.

3.6 Finite Element method

Idea of the Projection method: We don't solve the PDE exactly everywhere. Instead, we project it onto a finite dimensional subspace and solve it approximately, but in a mathematically optimal sense.

Let's take the Poisson equation as an example:

$$-\nabla^2 u = f \quad \text{in } \Omega, \quad u = 0 \text{ on } \partial\Omega.$$

We are looking for u in an infinite dimensional space, with all functions that vanish on the boundary and have square integrable derivatives:

$$V = \left\{ v \in H_0^1(\Omega) \mid \nabla v \in L^2(\Omega) \right\},$$

this is a Hilbert space (an infinite dimensional vector space equipped with an inner product).

The FEM is going to approximate u by a function u_h from a finite-dimensional subspace $V_h \subset V$ (spanned by basis functions over a defined mesh for this problem). In particular, We don't expect u_h to satisfy the PDE exactly everywhere. Instead, we require it to satisfy the PDE in an weak sense (averaged sense).

Start from the original PDE, we multiply both sides by a test function $v \in V$, and then integrate over the domain of Ω:

$$\int_\Omega -\left(\nabla^2 u\right) v dx = \int_\Omega f v dx.$$

Next, we can integrate by parts, to obatin:

$$\int_\Omega \nabla u \cdot \nabla v dx - \int_{\partial\Omega} v \frac{\partial u}{\partial n} ds = \int_\Omega f v dx.$$

Because $v = 0$ on the boundary, the boundary term vanishes. Therefore, We get the weak form to the original problem. Find $u \in V$ such that $a(u, v) = L(v)$ for all $v \in V$, where

$$\begin{aligned} a(u, v) &= \int_\Omega \nabla u \cdot \nabla v dx, \\ L(v) &= \int_\Omega f v dx. \end{aligned}$$

To discretize the weak formulation above, we will consider a finite dimensional subspace:

$$V_h = \operatorname{span}\left\{\phi_1, \phi_2, \ldots, \phi_N\right\}$$

where ϕ_i are basis functions. In particular, we can approximate u by a linear representation over the bases (there are many choices of basis set), the particular one of choice is Lagrangian basis (which gives you the nodal finite element method):

$$u_h = \sum_{i=1}^{N} U_i \ell_i,$$

where U_i is the corresponding coefficient of basis ℓ_i. Next, choose the test functions to be the same as basis function $v_h = \ell_j$ (so called standard Galerkin FEM), we derive:

$$a\left(u_h, \ell_j\right) = L\left(\ell_j\right), \quad \text{for } j = 1, \ldots, N,$$

which is equivalent to:

$$\sum_i U_i \int_\Omega \nabla \ell_i \cdot \nabla \ell_j dx = \int_\Omega f \ell_j dx.$$

This results a system of linear equations:

$$K\mathbf{U} = \mathbf{b},$$

where system matrix K's entry is given by:

$$K_{ij} = a\left(\ell_i, \ell_j\right) = \int_\Omega \nabla \ell_i \cdot \nabla \ell_j dx,$$

and

$$b_j = L\left(\ell_j\right) = \int_\Omega f \ell_j dx.$$

How do you calculate those integrals? Gaussian quadrature:)

Remark: Another way to interpret FEM is to find the best approximation $u_h \in V_h$ such that the residual is orthogonal to all test functions.

$$a\left(u - u_h, v_h\right) = 0, \quad \forall v_h \in V_h.$$

This is a Galerkin projection onto V_h.

Example 1: Solve the 1D Poisson equation using linear basis function.

$$-u'' = f, \quad \text{with } x \in (0,1), \text{ and BCs: } u(0) = u(1) = 0.$$

Multiply the equation by the test function $v \in H_0^1(0,1)$, and integrate by parts:

$$\int_0^1 u'(x)v'(x)dx = \int_0^1 f(x)v(x)dx.$$

Define the bilinear and linear forms:

$$a(u,v) = \int_0^1 u'(x)v'(x)dx, \quad L(v) = \int_0^1 f(x)v(x)dx.$$

Now the problem changes to find $u \in H_0^1(0,1)$ such that $a(u,v) = L(v)$, for all $v \in H_0^1(0,1)$.

Next, we partition $[0,1]$ into N elements:

$$0 = x_0 < x_1 < \cdots < x_N = 1,$$

with each elemental length of $h_e = x_{e+1} - x_e$ (Doesn't have to be the same for all elements). We approximate u and v in the finite element space:

$$V_h = \text{span}\left\{\phi_1, \phi_2, \ldots, \phi_{N-1}\right\},$$

where ϕ_j are the basis functions:

$$\phi_j\left(x_i\right) = \delta_{ij}, \quad \phi_j(0) = \phi_j(1) = 0.$$

The discrete problem is: Find $u_h(x) = \sum_{j=1}^{N-1} U_j \phi_j(x)$ such that:

$$\sum_{j=1}^{N-1} U_j a\left(\phi_j, \phi_i\right) = L\left(\phi_i\right), \quad i = 1, \ldots, N-1.$$

To construct the entries of the system matrices, consider the local linear basis functions. For calculation simplicity (for easy integration purpose (Guassian quadrature)),

we will map the physical coordinates $x \in [x_e, x_{e+1}]$ to standard coordinates $\xi \in [-1, 1]$ via the mapping:

$$x = x_c + \frac{h_e}{2}\xi, \quad x_c = \frac{x_e + x_{e+1}}{2}, \quad dx = \frac{h_e}{2}d\xi.$$

The basis functions and their derivatives in the reference domain $[-1, 1]$ are listed as follows.

$$\hat{\phi}_1(\xi) = \tfrac{1-\xi}{2}, \quad \hat{\phi}_2(\xi) = \tfrac{1+\xi}{2},$$
$$\tfrac{d\hat{\phi}_1}{d\xi} = -\tfrac{1}{2}, \qquad \tfrac{d\hat{\phi}_2}{d\xi} = \tfrac{1}{2}.$$

Once you map to the physical element, the corresponding physical basis functions are obtained by composition:

$$\phi_j(x) = \hat{\phi}_j(\xi(x)).$$

Therefore, by using the chain rule, we know the physical corresponding derivatives become:

$$\frac{d\phi_j}{dx} = \frac{d\hat{\phi}_j}{d\xi}\frac{d\xi}{dx} = \frac{2}{h_e}\frac{d\hat{\phi}_j}{d\xi} \quad \Rightarrow \quad \frac{d\phi_1}{dx} = -\frac{1}{h_e}, \quad \frac{d\phi_2}{dx} = \frac{1}{h_e}.$$

Next, let's calculate the element stiffness matrix $K_{ij}^{(e)}$:

$$K_{ij}^{(e)} = a\left(\phi_j, \phi_i\right)\big|_e = \int_{x_e}^{x_{e+1}} \frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx = \int_{-1}^{1} \frac{2}{h_e}\frac{d\hat{\phi}_i}{d\xi}\frac{2}{h_e}\frac{d\hat{\phi}_j}{d\xi}\frac{h_e}{2}d\xi = \frac{1}{h_e}\int_{-1}^{1} \frac{d\hat{\phi}_i}{d\xi}\frac{d\hat{\phi}_j}{d\xi}d\xi$$

Since $\frac{d\hat{\phi}_i}{d\xi}$ are constants, we can evaluate directly, and we have:

$$K^{(e)} = \frac{1}{h_e}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

For higher order basis functons, $\frac{d\hat{\phi}_i}{d\xi}$ won't be constants, so you will want to use Gaussian quadrature as follows:

$$K_{ij}^{(e)} \approx \frac{1}{h_e}\sum_{q=1}^{Q} w_q \frac{d\hat{\phi}_i}{d\xi}\left(\xi_q\right)\frac{d\hat{\phi}_j}{d\xi}\left(\xi_q\right),$$

where ξ_q, w_q are Gauss-Legendre nodes/weights (For example, 2 points quadrature gives you $\pm 1/\sqrt{3}, 1$).

Regarding the RHS (Elementary force load), we have:

$$f_i^{(e)} = L\left(\phi_i\right)\big|_e = \int_{x_e}^{x_{e+1}} f(x)\phi_i(x)dx = \int_{-1}^{1} f\left(x_c + \frac{h_e}{2}\xi\right)\hat{\phi}_i(\xi)\frac{h_e}{2}d\xi,$$

which can be calculated by using Gauss-Legendre quadrature:

$$f_i^{(e)} \approx \frac{h_e}{2}\sum_{q=1}^{Q} w_q f\left(x_c + \frac{h_e}{2}\xi_q\right)\hat{\phi}_i\left(\xi_q\right).$$

For the simpliest case, when $f(x)$ is a constant:

$$f^{(e)} = \frac{fh_e}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Now the things left to do is assemble the system matrix.

Example 2: Consider the domain is discretized by 3 elements.

We will have a total of 4 nodes, with the global node numbering as follows:

$$x_0 - - x_1 - - x_2 - - x_3$$

Each element contains the following intervals:

$$\begin{aligned} e_1 &:= [x_0, x_1], \\ e_2 &:= [x_1, x_2], \\ e_3 &:= [x_2, x_3]. \end{aligned}$$

With each element contributes a $K^{(e)}$ in the system matrix:

$$K^{(e)} = \frac{1}{h_e} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

So the global matrix before BCs enforcement looks like:

$$K = \begin{bmatrix} K_{00} & K_{01} & 0 & 0 \\ K_{10} & K_{11} & K_{12} & 0 \\ 0 & K_{21} & K_{22} & K_{23} \\ 0 & 0 & K_{32} & K_{33} \end{bmatrix} = \begin{bmatrix} \frac{1}{h_1} & -\frac{1}{h_1} & 0 & 0 \\ -\frac{1}{h_1} & \frac{1}{h_1} + \frac{1}{h_2} & -\frac{1}{h_2} & 0 \\ 0 & -\frac{1}{h_2} & \frac{1}{h_2} + \frac{1}{h_3} & -\frac{1}{h_3} \\ 0 & 0 & -\frac{1}{h_3} & \frac{1}{h_3} \end{bmatrix}.$$

This matrix arises by summing element contributions at shared nodes. For example, node x_1 receives the overlap of element e_1 and e_2.

Regarding the global force load RHS, we have:

$$\mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} + f_1^{(2)} \\ f_2^{(2)} + f_1^{(3)} \\ f_2^{(3)} \end{bmatrix} = \begin{bmatrix} \frac{h_1}{2} f(x_0) \\ \frac{h_1}{2} f(x_1) + \frac{h_2}{2} f(x_1) \\ \frac{h_2}{2} f(x_2) + \frac{h_3}{2} f(x_2) \\ \frac{h_3}{2} f(x_3) \end{bmatrix}.$$

Now we have the system of:

$$K\mathbf{U} = \mathbf{b}.$$

To enforce the BCs such as $u_0 = 0$, we can set $K_{00} = 1$ and $b_0 = 0$.

Example 3: Let's try to use quadratic basis.

Discretize with P_2 on a mesh $0 = x_0 < x_1 < \cdots < x_N = 1$ (elements $e = [x_e, x_{e+1}]$). We can choose the global indexing of $0, 1, 2, \ldots, 2N$, and the P_2 basis on the standard domain $[-1, 1]$ as below:

$$\hat{\phi}_1(\xi) = \frac{\xi(\xi - 1)}{2}, \quad \hat{\phi}_2(\xi) = 1 - \xi^2, \quad \hat{\phi}_3(\xi) = \frac{\xi(\xi + 1)}{2}, \quad \text{at } \xi = -1, 0, 1 \text{ respectively.}$$

with corresponding mapping:

$$x = x_c + \frac{h_e}{2}\xi, \quad \xi \in [-1, 1], \quad dx = \frac{h_e}{2} d\xi, \quad \frac{d}{dx} = \frac{2}{h_e} \frac{d}{d\xi}.$$

The corresponding derivatives are:

$$\hat{\phi}_1'(\xi) = \xi - \frac{1}{2}, \quad \hat{\phi}_2'(\xi) = -2\xi, \quad \hat{\phi}_3'(\xi) = \xi + \frac{1}{2}$$

with the physical derivatives can be obtained with chain rule:

$$\phi_i'(x) = \frac{2}{h_e}\hat{\phi}_i'(\xi).$$

The construction of element matrices are similar as linear basis, and the integral calculation can be done via Gaussian quadrature. For the stiffness elementary matrix $K_{ij}^{(e)}$:

$$K_{ij}^{(e)} = \int_{x_e}^{x_{e+1}} \phi_i'(x)\phi_j'(x)dx = \frac{2}{h_e}\int_{-1}^{1} \hat{\phi}_i'(\xi)\hat{\phi}_j'(\xi)d\xi = \frac{1}{3h_e}\begin{bmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{bmatrix}.$$

Remark: Notice that the integrand $\hat{\phi}_i'\hat{\phi}_j'$ is at most quadratic (why?). So 2 point Gauss provides exact integral, so you don't have to use more than 3 points for Gaussian quadrature.

For the RHS:

$$f_i^{(e)} = \int_{x_e}^{x_{e+1}} f(x)\phi_i(x)dx = \frac{h_e}{2}\int_{-1}^{1} f\left(x_c + \frac{h_e}{2}\xi\right)\hat{\phi}_i(\xi)d\xi \approx \frac{h_e}{2}\sum_{q=1}^{Q} w_q f\left(x_q\right)\hat{\phi}_i\left(\xi_q\right)$$

with Gauss points $\{\xi_q, w_q\}$ on $[-1,1]$ and $x_q = \frac{x_e+x_{e+1}}{2} + \frac{h_e}{2}\xi_q$. When f is constant, we have:

$$f^{(e)} = fh_e\begin{bmatrix} \frac{1}{6} \\ \frac{2}{3} \\ \frac{1}{6} \end{bmatrix}.$$

Global indices:

$0(\text{left})-1(\text{midpt of}\,[x_0,x_1]\,)-2(x_1)-3(\text{midpt}\,[x_1,x_2]\,)-4(x_2)-5(\text{midpt}\,[x_2,x_3]\,)-6(\text{right}).$

Given:

$$K^{(e)} = \frac{1}{3h}\begin{bmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{bmatrix}$$

We can assemble the global system matrix, by performing sum at shared DoFs.

$$K = \frac{1}{3h}\begin{bmatrix} 7 & -8 & 1 & 0 & 0 & 0 & 0 \\ -8 & 16 & -8 & 0 & 0 & 0 & 0 \\ 1 & -8 & 14 & -8 & 1 & 0 & 0 \\ 0 & 0 & -8 & 16 & -8 & 0 & 0 \\ 0 & 0 & 1 & -8 & 14 & -8 & 1 \\ 0 & 0 & 0 & 0 & -8 & 16 & -8 \\ 0 & 0 & 0 & 0 & 1 & -8 & 7 \end{bmatrix}.$$

For constant $f \equiv f$, each element load is

$$f^{(e)} = f_0 h[1/6, 2/3, 1/6]^T.$$

Thus the global load vector is:

$$\mathbf{b} = \begin{bmatrix} \frac{h}{6}f \\ \frac{2h}{3}f \\ \frac{h}{3}f \\ \frac{2h}{3}f \\ \frac{h}{3}f \\ \frac{2h}{3}f \\ \frac{h}{6}f \end{bmatrix}$$

Applying dirichlet BCs $u(0) = u(1) = 0$, we can remove rows/cols for DoFs 0 and 6 (this is another approach different from before, it only applies when you have homogeneous Dirichlet boundary condition), we have:

$$\frac{1}{3h}\begin{bmatrix} 16 & -8 & 0 & 0 & 0 \\ -8 & 14 & -8 & 1 & 0 \\ 0 & -8 & 16 & -8 & 0 \\ 0 & 1 & -8 & 14 & -8 \\ 0 & 0 & 0 & -8 & 16 \end{bmatrix}\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{bmatrix} = \begin{bmatrix} \frac{2h}{3}f \\ \frac{h}{3}f \\ \frac{2h}{3}f \\ \frac{h}{3}f \\ \frac{2h}{3}f \end{bmatrix}.$$

2D finite element

Let's consider the linear traigular element. A reference triangle is usually defined with vertices at:

$$(\xi,\eta)_1 = (0,0), \quad (\xi,\eta)_2 = (1,0), \quad (\xi,\eta)_3 = (0,1)$$

On this triangle, the linear (P1) basis functions are:

$$\begin{aligned} \hat{\phi}_1(\xi,\eta) &= 1 - \xi - \eta, \\ \hat{\phi}_2(\xi,\eta) &= \xi, \\ \hat{\phi}_3(\xi,\eta) &= \eta. \end{aligned}$$

These satisfy the Kronecker property:

$$\hat{\phi}_i\left(\xi_j,\eta_j\right) = \delta_{ij}, \quad i,j = 1,2,3,$$

and also form a partition of unity:

$$\hat{\phi}_1 + \hat{\phi}_2 + \hat{\phi}_3 = 1.$$

Let the physical triangular element have vertices $\left(x_i,y_i\right), i = 1,2,3$. We can map from the reference coordinates (ξ,η) to physical coordinates (x,y) via the mapping:

$$\begin{aligned} x(\xi,\eta) &= x_1\hat{\phi}_1 + x_2\hat{\phi}_2 + x_3\hat{\phi}_3 = x_1 + \left(x_2 - x_1\right)\xi + \left(x_3 - x_1\right)\eta, \\ y(\xi,\eta) &= y_1\hat{\phi}_1 + y_2\hat{\phi}_2 + y_3\hat{\phi}_3 = y_1 + \left(y_2 - y_1\right)\xi + \left(y_3 - y_1\right)\eta. \end{aligned}$$

Then, in the physical triangle, the basis functions are:

$$\phi_i(x,y) = a_i + b_i x + c_i y, \quad i = 1,2,3$$

with the coefficients are obtained from the vertex coordinates:

$$\begin{aligned} a_1 &= x_2y_3 - x_3y_2, \quad b_1 = y_2 - y_3, \quad c_1 = x_3 - x_2, \\ a_2 &= x_3y_1 - x_1y_3, \quad b_2 = y_3 - y_1, \quad c_2 = x_1 - x_3, \\ a_3 &= x_1y_2 - x_2y_1, \quad b_3 = y_1 - y_2, \quad c_3 = x_2 - x_1. \end{aligned}$$

The area of the triangle is:

$$2\Delta = x_1 (y_2 - y_3) + x_2 (y_3 - y_1) + x_3 (y_1 - y_2) .$$

The normalized physical shape functions are:

$$\phi_i(x, y) = \frac{1}{2\Delta} (a_i + b_i x + c_i y) , \quad i = 1, 2, 3,$$

Which satisfy:

$$\phi_i (x_j, y_j) = \delta_{ij}, \quad \phi_1 + \phi_2 + \phi_3 = 1.$$

Let's try to solve the 2D Poisson equation. Find $u : \Omega \to \mathbb{R}$ such that

$$-\nabla^2 u = f \quad \text{in } \Omega, \quad u = 0 \quad \text{on } \partial\Omega_D.$$

The weak form seek for $u \in V_0 = \left\{ v \in H^1(\Omega) : v|_{\partial\Omega_D} = 0 \right\}$ such that:

$$a(u, v) = (f, v), \quad \forall v \in V_0.$$

where

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v d\Omega, \quad (f, v) = \int_\Omega f v d\Omega.$$

Next, we first calculate the elementary stiffness (Here is 3×3 matrix):

$$K_{ij}^{(e)} = \int_{T_e} \nabla \phi_i \cdot \nabla \phi_j dA = \frac{1}{4\Delta_e} (b_i b_j + c_i c_j)$$

Since the gradients are constant within each triangle:

$$\nabla \phi_i = \begin{bmatrix} \partial \phi_i / \partial x \\ \partial \phi_i / \partial y \end{bmatrix} = \frac{1}{2\Delta} \begin{bmatrix} b_i \\ c_i \end{bmatrix}$$

Therefore, in matrix form we can write $K^{(e)}$ as:

$$K^{(e)} = \frac{1}{4\Delta_e} \begin{bmatrix} b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}^T \begin{bmatrix} b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} = \frac{1}{4\Delta_e} \begin{bmatrix} b_1^2 + c_1^2 & b_1 b_2 + c_1 c_2 & b_1 b_3 + c_1 c_3 \\ b_2 b_1 + c_2 c_1 & b_2^2 + c_2^2 & b_2 b_3 + c_2 c_3 \\ b_3 b_1 + c_3 c_1 & b_3 b_2 + c_3 c_2 & b_3^2 + c_3^2 \end{bmatrix}.$$

For simplicity, we assume a constant f on triangular element, a standard exact formula is:

$$F^{(e)} = \int_{T_e} f \phi dA = f_e \frac{\Delta_e}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

To assemble the global stiffness matrix A and force vector F, we do things similar to the 1D case:

$$K = \sum_e K^{(e)}, \quad F = \sum_e F^{(e)}.$$

4 Systems of Linear Equations

In this chapter we consider methods for solving a linear system of n equations in n variables. Such a system has the form

$$\begin{aligned} E_1: &\quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, \\ E_2: &\quad a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \\ &\qquad\qquad \vdots \\ E_n: &\quad a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n. \end{aligned}$$

Notations:

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

$$\text{Augmented matrix: } [A, \mathbf{b}] = \left[\begin{array}{cccc:c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right].$$

4.1 Gaussian elimination

The general Gaussian elimination procedure applied to the linear system:

$$\begin{aligned} E_1: &\quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ E_2: &\quad a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ &\qquad\qquad \vdots \\ E_n: &\quad a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{aligned}$$

is handled by first forming the augmented matrix $\tilde{A}$:

$$\tilde{A} = [A, \mathbf{b}] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & \cdots & a_{2n} & a_{2,n+1} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & a_{n,n+1} \end{bmatrix},$$

where A denotes the matrix formed by the coefficients. The entries in the $(n+1)$ st column are the values of $\mathbf{b}$; that is, $a_{i,n+1} = b_i$ for each $i = 1, 2, \cdots, n$.

Provided $a_{11} \neq 0$, we perform the operations corresponding to

$$(E_j - (a_{j1}/a_{11})\, E_1) \to (E_j) \qquad \text{for each } j = 2, 3, \ldots, n$$

to eliminate the coefficient of x_1 in each of these rows. Although the entries in rows $2, 3, \ldots, n$ are expected to change, for ease of notation we again denote the entry in the i th row and the j th column by a_{ij}. With this in mind, we follow a sequential procedure for $i = 2, 3, \ldots, n-1$ and perform the operation:

$$(E_j - (a_{ji}/a_{ii})\, E_i) \to (E_j) \qquad \text{for each } j = i+1, i+2, \ldots, n,$$

provided $a_{ii} \neq 0$. This eliminates all the a_{ij} entries below the a_{ii} entries for all values of $j < i$. The resulting matrix has the form:

$$\tilde{\tilde{A}} = \left[\begin{array}{ccccc|c} a_{11} & a_{12} & \cdots & a_{1n} & \vdots & a_{1,n+1} \\ 0 & a_{22} & \cdots & a_{2n} & \vdots & a_{2,n+1} \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & a_{nn} & \vdots & a_{n,n+1} \end{array}\right],$$

where, except in the first row, the values of a_{ij} are not expected to agree with those in the original matrix $\tilde{A}$. The matrix $\tilde{\tilde{A}}$ represents a linear system with the same solution set as the original system.

The new linear system is triangular,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= a_{1,n+1}, \\ a_{22}x_2 + \cdots + a_{2n}x_n &= a_{2,n+1}, \\ &\vdots \\ a_{nn}x_n &= a_{n,n+1}, \end{aligned}$$

so backward substitution can be performed. Solving the nth equation for x_n gives

$$x_n = \frac{a_{n,n+1}}{a_{nn}}.$$

Solving the $(n-1)$ st equation for x_{n-1} and using the known value for x_n yields:

$$x_{n-1} = \frac{a_{n-1,n+1} - a_{n-1,n}x_n}{a_{n-1,n-1}}.$$

Continuing this process, we obtain:

$$x_i = \frac{a_{i,n+1} - a_{i,n}x_n - a_{i,n-1}x_{n-1} - \cdots - a_{i,i+1}x_{i+1}}{a_{ii}} = \frac{a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}},$$

for each $i = n-1, n-2, \cdots, 2, 1$.

Example 1: Gaussian Elimination with row pivoting

Represent the linear system

$$\begin{aligned} E_1: &\quad x_1 - x_2 + 2x_3 - x_4 = -8, \\ E_2: &\quad 2x_1 - 2x_2 + 3x_3 - 3x_4 = -20, \\ E_3: &\quad x_1 + x_2 + x_3 = -2, \\ E_4: &\quad x_1 - x_2 + 4x_3 + 3x_4 = 4, \end{aligned}$$

as an augmented matrix and use Gaussian Elimination to find its solution. Solution The augmented matrix is

$$\tilde{A} = \tilde{A}^{(1)} = \left[\begin{array}{cccc|c} 1 & -1 & 2 & -1 & \vdots & -8 \\ 2 & -2 & 3 & -3 & \vdots & -20 \\ 1 & 1 & 1 & 0 & \vdots & -2 \\ 1 & -1 & 4 & 3 & \vdots & 4 \end{array}\right].$$

Performing the operations

$$(E_2 - 2E_1) \to (E_2), (E_3 - E_1) \to (E_3), \quad \text{and} \quad (E_4 - E_1) \to (E_4),$$

gives

$$\tilde{A}^{(2)} = \begin{bmatrix} 1 & -1 & 2 & -1 & -8 \\ 0 & 0 & -1 & -1 & -4 \\ 0 & 2 & -1 & 1 & 6 \\ 0 & 0 & 2 & 4 & 12 \end{bmatrix}$$

The diagonal entry $a_{22}^{(2)}$, called the pivot element, is 0 , so the procedure cannot continue in its present form. But operations $(E_i) \leftrightarrow (E_j)$ are permitted, so a search is made of the elements $a_{32}^{(2)}$ and $a_{42}^{(2)}$ for the first nonzero element. Since $a_{32}^{(2)} \neq 0$, the operation $(E_2) \leftrightarrow (E_3)$ is performed to obtain a new matrix,

$$\tilde{A}^{(2)'} = \begin{bmatrix} 1 & -1 & 2 & -1 & -8 \\ 0 & 2 & -1 & 1 & 6 \\ 0 & 0 & -1 & -1 & -4 \\ 0 & 0 & 2 & 4 & 12 \end{bmatrix}.$$

Since x_2 is already eliminated from E_3 and E_4, $\tilde{A}^{(3)}$ will be $\tilde{A}^{(2)'}$, and the computations continue with the operation $(E_4 + 2E_3) \to (E_4)$, giving

$$\tilde{A}^{(4)} = \begin{bmatrix} 1 & -1 & 2 & -1 & -8 \\ 0 & 2 & -1 & 1 & 6 \\ 0 & 0 & -1 & -1 & -4 \\ 0 & 0 & 0 & 2 & 4 \end{bmatrix}.$$

Finally, the matrix is converted back into a linear system that has a solution equivalent to the solution of the original system and the backward substitution is applied:

$$\begin{aligned} x_4 &= \frac{4}{2} = 2 \\ x_3 &= \frac{[-4 - (-1)x_4]}{-1} = 2 \\ x_2 &= \frac{[6 - x_4 - (-1)x_3]}{2} = 3 \\ x_1 &= \frac{[-8 - (-1)x_4 - 2x_3 - (-1)x_2]}{1} = -7 \end{aligned}$$

Algorithm of Gaussian Elimination

To solve the $n \times n$ linear system:

$$\begin{aligned} E_1 &: \quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = a_{1,n+1} \\ E_2 &: \quad a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = a_{2,n+1} \\ &\vdots \\ E_n &: \quad a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = a_{n,n+1} \end{aligned}$$

INPUT number of unknowns and equations n; augmented matrix $A = [a_{ij}]$, where $1 \leq i \leq n$ and $1 \leq j \leq n + 1$.

OUTPUT solution $x_1, x_2, \ldots, x_n$ or message that the linear system has no unique solution.

Step 1 For $i = 1, \ldots, n-1$ do Steps 2-4. (Elimination process.)

Step 2 Let p be the smallest integer with $i \leq p \leq n$ and $a_{pi} \neq 0$. If no integer p can be found then OUTPUT ('no unique solution exists'); STOP.

Step 3 If $p \neq i$ then perform $(E_p) \leftrightarrow (E_i)$.

Step 4 For $j = i+1, \ldots, n$ do Steps 5 and 6.

Step 5 Set $m_{ji} = a_{ji}/a_{ii}$.

Step 6 Perform $(E_j - m_{ji}E_i) \rightarrow (E_j)$;

Step 7 If $a_{nn} = 0$ then OUTPUT ('no unique solution exists'); STOP.

Step 8 Set $x_n = a_{n,n+1}/a_{nn}$. (Start backward substitution.)

Step 9 For $i = n-1, \ldots, 1$ set $x_i = \left[a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij}x_j\right]/a_{ii}$.

Step 10 OUTPUT $(x_1, \ldots, x_n)$; (Procedure completed successfully.)

STOP.

Remarks: Step 2 is a pivoting procedure. It looks for a pivot element in column i, such that starts from row i and going downward, looking for the first row p that has a_{pi} is nonzero. If it can't be found, then no solution (matrix is singular).

What Step 3 does it that once the pivoting element is found, we do the row pivoting.

Parallelization?: Steps 4 can be parallelized. Namely, the outer loop over rows $j = i+1 \cdots n$ can be parallelized. Each thread updates one row.

Matlab implementation:

```
%Gaussian elimination
% Step 1: For i = 1, ..., n-1 do Steps 24.
   for i = 1:n-1
      % Step 2: find smallest p in [i,n] with a(p,i) ~= 0; else no unique
          solution.
      rel = find(a(i:n, i) ~= 0, 1, 'first'); % first nonzero from row i
          downward
      if isempty(rel)
         disp('no unique solution exists');
         x = [];
         return
      end
      p = i + rel - 1;

      % Step 3: swap rows E_p <-> E_i, if p ~= i.
      if p ~= i
         tmp = a(i, :);
         a(i, :) = a(p, :);
         a(p, :) = tmp;
      end

      % Step 4: For j = i+1, ..., n do Steps 5 and 6.
      for j = i+1:n
         % Step 5: m_{ji} = a(j,i) / a(i,i).
         mji = a(j, i) / a(i, i);
```

```
24
25          % Step 6: (E_j - m_{ji} E_i) -> E_j (apply to all columns i..n+1)
26          a(j, i:m) = a(j, i:m) - mji * a(i, i:m);
27       end
28    end
29
30    % Step 7: If a(n,n) = 0 then no unique solution.
31    if a(n, n) == 0
32       disp('no unique solution');
33       x = [];
34       return
35    end
36
37    x = zeros(n, 1);
38
39    % Step 8: x_n = a(n, n+1) / a(n, n)
40    x(n) = a(n, n+1) / a(n, n);
41
42    % Step 9: For i = n-1, ..., 1:
43    for i = n-1:-1:1
44       s = 0;
45       for j = i+1:n
46          s = s + a(i, j) * x(j);
47       end
48       x(i) = (a(i, n+1) - s) / a(i, i);
49    end
```

Does Gaussian elimination always gives you correct answers?

Think about finite precision arithmetic, rounding error... It can be unstable if system has a large condition number, and the elimination process gives large values or division involving small values, and so on..

Operation Counts

Both the amount of time required to complete the calculations and the subsequent round-off error depend on the number of floating-point arithmetic operations needed to solve a routine problem. In general, the amount of time required to perform a multiplication or division on a computer is approximately the same and is considerably greater than that required to perform an addition or subtraction. The actual differences in execution time, however, depend on the particular computing system.

No arithmetic operations are performed until Steps 5 and 6 in the algorithm.

"**Step 5: Set** $m_{ji} = a_{ji}/a_{ii}$."

Step 5 requires that ($n - i$) divisions be performed (all $a_j i$ entries in the rows below i-th row).

"**Step 6: Perform** $(E_j - m_{ji}E_i) \rightarrow (E_j)$"

The replacement of the equation E_j by $(E_j - m_{ji}E_i)$ in Step 6 requires that m_{ji} be multiplied by each term in E_i, resulting in a total of $(n - i)(n - i + 1)$ multiplications. (Consider that each equation E_j typically has coefficients $a_{j,i}, a_{j,i+1}, \ldots, a_{j,n}$, there are $(n - i + 1)$ terms per equation.)

After this is completed, each term of the resulting equation is subtracted from the corresponding term in E_j. This requires $(n-i)(n-i+1)$ subtractions.

Therefore, for each $i = 1, 2, \ldots, n-1$, the operations required in Steps 5 and 6 are as follows.

1. Multiplications/divisions:

$$(n-i) + (n-i)(n-i+1) = (n-i)(n-i+2).$$

2. Additions/subtractions:

$$(n-i)(n-i+1).$$

The total number of operations required by Steps 5 and 6 is obtained by summing the operation counts for each i. Recalling from calculus that:

$$\sum_{j=1}^{m} 1 = m, \quad \sum_{j=1}^{m} j = \frac{m(m+1)}{2}, \quad \text{and} \quad \sum_{j=1}^{m} j^2 = \frac{m(m+1)(2m+1)}{6}$$

We have the following operation counts.

1. Multiplications/divisions:

$$\begin{aligned}\sum_{i=1}^{n-1}(n-i)(n-i+2) &= \sum_{i=1}^{n-1}(n-i)^2 + 2\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i^2 + 2\sum_{i=1}^{n-1} i \\ &= \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2} = \frac{2n^3+3n^2-5n}{6}.\end{aligned}$$

2. Additions/subtractions:

$$\begin{aligned}\sum_{i=1}^{n-1}(n-i)(n-i+1) &= \sum_{i=1}^{n-1}(n-i)^2 + \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} = \frac{n^3-n}{3}.\end{aligned}$$

The only other steps in Algorithm on page 76 that involve arithmetic operations are those required for backward substitution, Steps 8 and 9.

Step 8: Set $x_n = a_{n,n+1}/a_{nn}$. (Start backward substitution.)

Step 8 requires one division.

Step 9 For $i = n-1, \ldots, 1$ set $x_i = \left[a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij}x_j\right]/a_{ii}$.

Step 9 requires ($n-i$) multiplications and ($n-i-1$) additions for each summation term and then one subtraction and one division. The total number of operations in Steps 8 and 9 is as follows.

1. Multiplications/divisions:

$$\begin{aligned}1 + \sum_{i=1}^{n-1}((n-i)+1) &= 1 + \left(\sum_{i=1}^{n-1}(n-i)\right) + n - 1 \\ 1 = n + \sum_{i=1}^{n-1}(n-i) &= n + \sum_{i=1}^{n-1} i = \frac{n^2+n}{2}.\end{aligned}$$

2. Additions/subtractions:

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

The total number of arithmetic operations in Algorithm of Gaussian elimination is, therefore:

1. Multiplications/divisions:

$$\frac{2n^3+3n^2-5n}{6} + \frac{n^2+n}{2} = \frac{n^3}{3} + n^2 - \frac{n}{3}.$$

2. Additions/subtractions:

$$\frac{n^3-n}{3} + \frac{n^2-n}{2} = \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}.$$

For large n, the total number of multiplications and divisions is approximately $n^3/3$, as is the total number of additions and subtractions. Thus the amount of computation and the time required increases with n in proportion to n^3, as shown in the table below.

n	Multiplications/Divisions	Additions/Subtractions
3	17	11
10	430	375
50	44,150	42,875
100	343,300	338,250

Memory requirement

If the system is dense (no zero pattern exploited), for the augmented matrix $[A \mid b]$, which is an $n \times (n+1)$ array, the memory size is $= n^2 + n$.

Next, let's check the memory contents during elimination. At any stage i of Gaussian elimination, the algorithm needs to store:

1. The coefficient matrix $A = [a_{ij}]$ requires n^2 entries (if fully dense).
2. The right-hand side vector b, has n entries.
3. The multipliers $m_{ji} = \frac{a_{ji}}{a_{ii}}$, requires temporarily stored for $j = i+1, \ldots, n$.
4. Optionally, pivot information (indices) if partial pivoting is used.

In summary, we have:

Component	Memory Size (numbers)	Comment
Coefficient matrix A	n^2	main storage
RHS b	n	augmented column
Multiplier m_{ii}	1 at a time	negligible
Pivot indices	n integers	negligible
Total	$\approx n^2 + n$	dominated by matrix storage

In bytes, assuming double precision (8 bytes per number), the memory size is $= 8n(n+1)$ bytes.

n	Memory (approx)
100	0.08 MB
1,000	8 MB
10,000	800 MB

4.2 LU factorizations

We have seen that Gaussian elimination applied to an arbitrary linear system $A\mathbf{x} = \mathbf{b}$ requires $O(n^3/3)$ arithmetic operations to determine $\mathbf{x}$. However, to solve a linear system that involves an upper-triangular system requires only backward substitution, which takes $O(n^2)$ operations. The number of operations required to solve a lower-triangular systems is similar.

Suppose that A has been factored into the triangular form $A = LU$, where L is lower triangular and U is upper triangular. Then we can solve for $\mathbf{x}$ more easily by using a two-step process. First, we let $\mathbf{y} = U\mathbf{x}$ and solve the lower triangular system $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$. Since L is triangular, determining $\mathbf{y}$ from this equation requires only $O(n^2)$ operations. Once $\mathbf{y}$ is known, the upper triangular system $U\mathbf{x} = \mathbf{y}$ requires only an additional $O(n^2)$ operations to determine the solution $\mathbf{x}$. Therefore, solving a linear system $A\mathbf{x} = \mathbf{b}$ in factored form means that the number of operations needed is reduced from $O(n^3/3)$ to $O(2n^2)$.

Compare the approximate number of operations required to determine the solution to a linear system using a technique requiring $O(n^3/3)$ operations and one requiring $O(2n^2)$ when $n = 20, n = 100$, and $n = 1000$.

n	$n^3/3$	$2n^2$	% Reduction
10	$3.\overline{3} \times 10^2$	2×10^2	40
100	$3.\overline{3} \times 10^5$	2×10^4	94
1000	$3.\overline{3} \times 10^8$	2×10^6	99.4

As the example illustrates, the reduction factor increases dramatically with the size of the matrix. Not surprisingly, the reductions from the factorization come at a cost; determining the specific matrices L and U requires $O(n^3/3)$ operations.

How to construct matrix U

To see which matrices have an LU factorization and to find how it is determined, first suppose that Gaussian elimination can be performed on the system $A\mathbf{x} = \mathbf{b}$ without row interchanges. This is equivalent to having nonzero pivot elements $a_{ii}^{(i)}$, for each $i = 1, 2, \ldots, n$.

The first step in the Gaussian elimination process consists of performing, for each $j = 2, 3, \ldots, n$, the operations

$$(E_j - m_{j,1}E_1) \to (E_j), \quad \text{where} \quad m_{j,1} = \frac{a_{j1}^{(1)}}{a_{11}^{(1)}}.$$

These operations transform the system into one in which all the entries in the first column below the diagonal are zero.

The system of operations $(E_j - m_{j,1}E_1) \to (E_j)$ can be viewed in another way. It is simultaneously accomplished by multiplying the original matrix A on the left by the matrix

$$M^{(1)} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ -m_{21} & 1 & \ddots & \ddots & & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ -m_{n1} & 0 & \cdots & \cdots & 0 & 1 \end{bmatrix}.$$

This is called the first Gaussian transformation matrix. We denote the product of this matrix with A by $A^{(2)}$ and with $\mathbf{b}$ by $\mathbf{b}^{(2)}$, so

$$A^{(2)}\mathbf{x} = M^{(1)}A\mathbf{x} = M^{(1)}\mathbf{b} = \mathbf{b}^{(2)}.$$

In a similar manner we construct $M^{(2)}$, **the identity matrix with the entries below the diagonal in the second column replaced by the negatives of the multipliers**

$$m_{j,2} = \frac{a_{j2}^{(2)}}{a_{22}^{(2)}}.$$

The product of this matrix with $A^{(2)}$ has zeros below the diagonal in the first two columns, and we let

$$A^{(3)}\mathbf{x} = M^{(2)}A^{(2)}\mathbf{x} = M^{(2)}M^{(1)}A\mathbf{x} = M^{(2)}M^{(1)}\mathbf{b} = \mathbf{b}^{(3)}.$$

In general, with $A^{(k)}\mathbf{x} = \mathbf{b}^{(k)}$ already formed, multiply by the $\boldsymbol{k}$ th Gaussian transformation matrix

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

to obtain

$$A^{(k+1)}\mathbf{x} = M^{(k)}A^{(k)}\mathbf{x} = M^{(k)}\cdots M^{(1)}A\mathbf{x} = M^{(k)}\mathbf{b}^{(k)} = \mathbf{b}^{(k+1)} = M^{(k)}\cdots M^{(1)}\mathbf{b}.$$

The process ends with the formation of $A^{(n)}\mathbf{x} = \mathbf{b}^{(n)}$, where $A^{(n)}$ is the upper triangular matrix

$$A^{(n)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ddots & \ddots & \vdots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & a_{n-1,n}^{(n-1)} \\ 0 & \cdots & \cdots & \cdots & : & a_{nn}^{(n)} \end{bmatrix},$$

given by

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

This process forms the $U = A^{(n)}$ portion of the matrix factorization $A = LU$.

How to construct matrix L

To determine the complementary lower triangular matrix L, first recall the multiplication of $A^{(k)}\mathbf{x} = \mathbf{b}^{(k)}$ by the Gaussian transformation of $M^{(k)}$:

$$A^{(k+1)}\mathbf{x} = M^{(k)}A^{(k)}\mathbf{x} = M^{(k)}\mathbf{b}^{(k)} = \mathbf{b}^{(k+1)},$$

where $M^{(k)}$ generates the row operations

$$(E_j - m_{j,k}E_k) \to (E_j), \quad \text{for } j = k+1, \ldots, n.$$

To reverse the effects of this transformation and return to $A^{(k)}$ requires that the operations $(E_j + m_{j,k}E_k) \to (E_j)$ be performed for each $j = k+1, \ldots, n$. This is equivalent to multiplying by the inverse of the matrix $M^{(k)}$, the matrix

$$L^{(k)} = [M^{(k)}]^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ 0 & 0 & \cdots & m_{k+1,k} & \cdots & 0 \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & m_{n,k} & \cdots & 1 \end{bmatrix}.$$

The lower-triangular matrix L in the factorization of A, then, is the product of the matrices $L^{(k)}$:

$$L = L^{(1)}L^{(2)}\cdots L^{(n-1)} = \begin{bmatrix} 1 & & 0 & \cdots & \cdots & \cdots & \cdots \\ m_{21} & & 1 & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \vdots \\ \dot{m}_{n1} & \cdots & \cdots & \cdots & m_{n,n-1} & \ddots & 1 \end{bmatrix},$$

since the product of L with the upper-triangular matrix $U = M^{(n-1)}\cdots M^{(2)}M^{(1)}A$ gives

$$\begin{aligned} LU &= L^{(1)}L^{(2)}\cdots L^{(n-3)}L^{(n-2)}L^{(n-1)} \cdot M^{(n-1)}M^{(n-2)}M^{(n-3)}\cdots M^{(2)}M^{(1)}A \\ &= \left[M^{(1)}\right]^{-1}\left[M^{(2)}\right]^{-1}\cdots\left[M^{(n-2)}\right]^{-1}\left[M^{(n-1)}\right]^{-1} \cdot M^{(n-1)}M^{(n-2)}\cdots M^{(2)}M^{(1)}A = A \end{aligned}$$

Example 2:

(a) Determine the LU factorization for matrix A in the linear system $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 1 & 1 & 0 & 3 \\ 2 & 1 & -1 & 1 \\ 3 & -1 & -1 & 2 \\ -1 & 2 & 3 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ -3 \\ 4 \end{bmatrix}.$$

(b) Then use the factorization to solve the system

$$\begin{aligned} x_1 + x_2 + 3x_4 &= 8, \\ 2x_1 + x_2 - x_3 + x_4 &= 7, \\ 3x_1 - x_2 - x_3 + 2x_4 &= 14, \\ -x_1 + 2x_2 + 3x_3 - x_4 &= -7. \end{aligned}$$

Solution (a) The original system undergoes a sequence of operations $(E_2 - 2E_1) \to (E_2), (E_3 - 3E_1) \to (E_3), (E_4 - (-1)E_1) \to (E_4), (E_3 - 4E_2) \to (E_3), (E_4 - (-3)E_2) \to (E_4)$ converts the system to the triangular system:

$$\begin{aligned} x_1 + x_2 + 3x_4 &= 4, \\ -x_2 - x_3 - 5x_4 &= -7, \\ 3x_3 + 13x_4 &= 13, \\ -13x_4 &= -13. \end{aligned}$$

The multipliers m_{ij} and the upper triangular matrix produce the factorization

$$A = \begin{bmatrix} 1 & 1 & 0 & 3 \\ 2 & 1 & -1 & 1 \\ 3 & -1 & -1 & 2 \\ -1 & 2 & 3 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ -1 & -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 3 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 3 & 13 \\ 0 & 0 & 0 & -13 \end{bmatrix} = LU.$$

(b) To solve

$$A\mathbf{x} = LU\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ -1 & -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 3 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 3 & 13 \\ 0 & 0 & 0 & -13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 8 \\ 7 \\ 14 \\ -7 \end{bmatrix},$$

we first introduce the substitution $\mathbf{y} = U\mathbf{x}$. Then $\mathbf{b} = L(U\mathbf{x}) = L\mathbf{y}$. That is,

$$L\mathbf{y} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ -1 & -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 8 \\ 7 \\ 14 \\ -7 \end{bmatrix}.$$

This system is solved for $\mathbf{y}$ by a simple forward-substitution process:

$$\begin{aligned} y_1 &= 8; \\ 2y_1 + y_2 = 7, &\quad \Longrightarrow y_2 = 7 - 2y_1 = -9; \\ 3y_1 + 4y_2 + y_3 = 14, &\quad \Longrightarrow y_3 = 14 - 3y_1 - 4y_2 = 26; \\ -y_1 - 3y_2 + y_4 = -7, &\quad \Longrightarrow y_4 = -7 + y_1 + 3y_2 = -26. \end{aligned}$$

We then solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$, the solution of the original system; that is,

$$\begin{bmatrix} 1 & 1 & 0 & 3 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 3 & 13 \\ 0 & 0 & 0 & -13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 8 \\ -9 \\ 26 \\ -26 \end{bmatrix}.$$

Using backward substitution we obtain $x_4 = 2, x_3 = 0, x_2 = -1, x_1 = 3$.

LU factorization algorithm

To factor the $n \times n$ matrix $A = [a_{ij}]$ into the product of the lower triangular matrix $L = [l_{ij}]$ and the upper triangular matrix $U = [u_{ij}]$. Namely, $A = LU$, where the main diagonal of either L or U consists of all ones.

INPUT dimension n; the entries a_{ij}, $1 \le i, j \le n$ of A.

The diagonal $l_{11} = \cdots = l_{nn} = 1$ of L **OR** the diagonal $u_{11} = \cdots = u_{nn} = 1$ of U.

OUTPUT the entries l_{ij}, $1 \le j \le i$, $1 \le i \le n$ of L and the entries u_{ij}, $i \le j \le n$, $1 \le i \le n$ of U.

Step 1. Select l_{11} and u_{11} satisfying $l_{11}u_{11} = a_{11}$.

If $l_{11}u_{11} = 0$ then OUTPUT (Factorization impossible); STOP. (Or pivoting is needed first.)

Step 2. For $j = 2, \ldots, n$ set

$$u_{1j} = a_{1j}/l_{11} \quad \text{(First row of } U\text{)},$$

$$l_{j1} = a_{j1}/u_{11} \quad \text{(First column of } L\text{)}.$$

(Simply matching the 1st row and 1st column of A).

Step 3. For $i = 2, \ldots, n-1$ do Steps 4 and 5, to compute the next diagonal, row and column entries.

Step 4. Select l_{ii} and u_{ii} satisfying

$$l_{ii}u_{ii} = a_{ii} - \sum_{k=1}^{i-1} l_{ik}u_{ki}.$$

If $l_{ii}u_{ii} = 0$ then OUTPUT (Factorization impossible); STOP.

Step 5. For $j = i+1, \ldots, n$ set

$$u_{ij} = \frac{1}{l_{ii}}\left[a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\right] \quad \text{To compute the rest terms of the } i-th \text{ row of } U\text{)},$$

$$l_{ji} = \frac{1}{u_{ii}}\left[a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki}\right] \quad \text{To compute the rest terms of } i-th \text{ column of } L\text{)}.$$

Step 6. Select l_{nn} and u_{nn} satisfying

$$l_{nn}u_{nn} = a_{nn} - \sum_{k=1}^{n-1} l_{nk}u_{kn}.$$

(Note: If $l_{nn}u_{nn} = 0$, then $A = LU$ but A is singular.)

Step 7. OUTPUT (l_{ij} for $j = 1, \ldots, i$ and $i = 1, \ldots, n$);
OUTPUT (u_{ij} for $j = i, \ldots, n$ and $i = 1, \ldots, n$);
STOP.

Remark:

1. Every element a_{ij} of A satisfies:

$$a_{ij} = \sum_{k=1}^{n} l_{ik}u_{kj}.$$

2. You must choose which matrix has ones on its diagonal. Either $l_{11} = l_{22} = \cdots = l_{nn} = 1$, the Doolittle form (unit diagonal in L), or $u_{11} = u_{22} = \cdots = u_{nn} = 1$, the Crout form (unit diagonal in U).

Example 2:

Let

$$A = \begin{bmatrix} 4 & 8 & 12 \\ 2 & 7 & 7 \\ 6 & 9 & 21 \end{bmatrix}, \quad n = 3,$$

Perform the LU factorization without pivoting, and choose the Doolittle form, i.e. L has ones on its diagonal.

$$l_{11} = l_{22} = l_{33} = 1.$$

Determine all other l_{ij}, u_{ij}.

Step 1: Select $l_{11} = 1$. Compute u_{11} from

$$l_{11}u_{11} = a_{11} = 4 \Longrightarrow u_{11} = 4$$

Since $l_{11}u_{11} \neq 0$, continue.

Step 2: For $j = 2, 3$:

$$u_{1j} = \frac{a_{1j}}{l_{11}} = a_{1j} \Rightarrow u_{12} = 8, u_{13} = 12$$

For $j = 2, 3$:

$$l_{j1} = \frac{a_{j1}}{u_{11}} = \begin{cases} l_{21} = 2/4 = 0.5 \\ l_{31} = 6/4 = 1.5 \end{cases}$$

So far:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 1.5 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 8 & 12 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Step 3-5: (loop for $i = 2$)

Step 4: Compute u_{22}

$$l_{22}u_{22} = a_{22} - \sum_{k=1}^{1} l_{2k}u_{k2} = 7 - (0.5)(8) = 7 - 4 = 3.$$

Since $l_{22} = 1$, we get $u_{22} = 3$.

Step 5: For $j = 3$,

$$u_{23} = \frac{1}{l_{22}} \left[a_{23} - l_{21}u_{13}\right] = (7 - 0.5 \times 12) = 7 - 6 = 1.$$

Then compute l_{32}.

$$l_{32} = \frac{1}{u_{22}} \left[a_{32} - l_{31}u_{12}\right] = \frac{1}{3}(9 - 1.5 \times 8) = \frac{1}{3}(9 - 12) = -1.$$

Now we have:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 1.5 & -1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 8 & 12 \\ 0 & 3 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Step 6($i = n = 3$):

$$l_{33}u_{33} = a_{33} - \sum_{k=1}^{2} l_{3k}u_{k3}.$$

Compute

$$a_{33} - (l_{31}u_{13} + l_{32}u_{23}) = 21 - (1.5 \times 12 + (-1) \times 1) = 21 - (18 - 1) = 21 - 17 = 4.$$

With $l_{33} = 1$, we get $u_{33} = 4$.

Therefore, we have:

$$LU = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 1.5 & -1 & 1 \end{bmatrix} \begin{bmatrix} 4 & 8 & 12 \\ 0 & 3 & 1 \\ 0 & 0 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 \\ 2 & 7 & 7 \\ 6 & 9 & 21 \end{bmatrix} = A.$$

When LU is useful?

Solving $A\mathbf{x} = \mathbf{b}$ using LU takes:

1. LU decomposition: $O(n^3)$
2. Forward substitution (Ly=b): $O(n^2)$
3. Backward substitution (Ux = y) : $O(n^2)$

LU very efficient for repeated solves! Namely, if you have some system like $Ax^{N+1} = f(x^n)$. If the matrix A stays the same at each time step, then:

1. Compute LU once:

$$A = LU$$

which costs about $\frac{2}{3}n^3$ flops.

2. At each time step n, we solve

$$LX^{n+1} = f(X^n)$$

by two steps: forward substitution $Ly = f(X^n)$, and then a backward substitution: $UX^{n+1} = y$. Since each substitution step costs only $O(n^2)$. So, instead of recomputing the full factorization every time, you reuse L and U and save computational time.

4.3 Permutation

In the previous discussion we assumed that $A\mathbf{x} = \mathbf{b}$ can be solved using Gaussian elimination without row interchanges. From a practical standpoint, this method is only useful when row interchanges are not required to control the round-off error resulting from the use of finite-digit arithmetic. But on computer, numbers are always rounded at every step. If a pivot element is very small, dividing by it can cause large numerical errors. So modifications must be made when row interchanges are required. Below, We will introduce a class of matrices that are used to rearrange, or permute, rows of a given matrix.

An $n \times n$ permutation matrix $P = [p_{ij}]$ is a matrix obtained by rearranging the rows of I_n, the identity matrix. This gives a matrix with precisely one nonzero entry in each row and in each column, and each nonzero entry is with value of 1. For example, the matrix P given below:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

is a 3×3 permutation matrix, for any 3×3 matrix A, multiplying on the left by P has the effect of interchanging the second and third rows of A. Namely,

$$PA = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}.$$

Similarly, multiplying A on the right by P interchanges the second and third columns of A. Think about P is a matrix obatined by rearranging the rows (or columns) of the identity matrix I!

Two useful properties:

1. A permutation is just a reordering of rows which doesn't lose any information. So, there's always a way to undo it (reorder the rows back). That's what P^{-1} does, it reverses the permutation. In other words:

$$P^{-1}P = I.$$

2. P is made by taking the rows of the identity matrix and reordering them. So taking the transpose of P turns its rows into columns. That's the same as reordering the columns of I in the same way. **Notice that reordering columns in that way is exactly the inverse operation of reordering rows!** For example,

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad P^T = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notice here that $P^T = P$, and we always have:

$$P^T P = I = PP^T$$

which implies $P^T = P^{-1}$.

When solving a linear system

$$A\mathbf{x} = \mathbf{b}$$

Gaussian elimination eliminates unknowns step by step, but sometimes you must swap rows to avoid dividing by zero or by a very small number (which causes numerical instability). Instead of performing swaps during elimination, we could perform them beforehand, all at once, by reordering the equations in advance. Mathematically, that reordering is achieved by multiplying the system on the left by a permutation matrix P

$$PA\mathbf{x} = P\mathbf{b}.$$

Because this reordered system can be solved without pivoting, the reordered matrix PA can be factored as:

$$PA = LU,$$

where L is lower triangular and U is upper triangular. Now we can isolate A :

$$A = P^{-1}LU.$$

And since permutation matrices are orthogonal, meaning $P^{-1} = P^T$, we can write:

$$A = \left(P^T L\right) U.$$

When you reorder the rows of a lower triangular matrix, the result is generally not lower triangular anymore. So $P^T L$ loses that simple triangular structure unless $P = I$ (no permutation needed). Thus, $A = \left(P^T L\right) U$ is a valid factorization, but it's not a standard LU factorization anymore! As the L factor isn't triangular matrix. That's why, in practice, we prefer to keep the form:

$$PA = LU,$$

instead of moving P to the other side.

Example 1: Determine a factorization in the form $A = (P^t L)\, U$ for the matrix:

$$A = \begin{bmatrix} 0 & 0 & -1 & 1 \\ 1 & 1 & -1 & 2 \\ -1 & -1 & 2 & 0 \\ 1 & 2 & 0 & 2 \end{bmatrix}.$$

The matrix A cannot have an LU factorization because $a_{11} = 0$. However, using the row interchange $(E_1) \leftrightarrow (E_2)$, followed by $(E_3 + E_1) \rightarrow (E_3)$ and $(E_4 - E_1) \rightarrow (E_4)$, produces:

$$\begin{bmatrix} 1 & 1 & -1 & 2 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Then the row interchange $(E_2) \leftrightarrow (E_4)$, followed by $(E_4 + E_3) \rightarrow (E_4)$, gives the matrix:

$$U = \begin{bmatrix} 1 & 1 & -1 & 2 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{bmatrix}.$$

The permutation matrix associated with the row interchanges: $(E_1) \leftrightarrow (E_2)$ and $(E_2) \leftrightarrow (E_4)$ is:

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

and

$$PA = \begin{bmatrix} 1 & 1 & -1 & 2 \\ 1 & 2 & 0 & 2 \\ -1 & -1 & 2 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}.$$

Gaussian elimination is performed on PA using the same operations as on A, except without the row interchanges. That is, $(E_2 - E_1) \rightarrow (E_2)\,, (E_3 + E_1) \rightarrow (E_3)$, followed by $(E_4 + E_3) \rightarrow (E_4)$. The nonzero multipliers for PA are consequently,

$$m_{21} = 1, \quad m_{31} = -1, \quad \text{and} \quad m_{43} = -1,$$

and the LU factorization of PA is:

$$PA = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 & 2 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{bmatrix} = LU.$$

Multiplying by $P^{-1} = P^t$ produces the factorization:

$$A = P^{-1}(LU) = P^t(LU) = \left(P^t L\right) U = \begin{bmatrix} 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 & 2 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{bmatrix}.$$

4.4 Cholesky factorization

To factor a **symmetric positive definite** $n \times n$ matrix $A = [a_{ij}]$ into the product of a lower triangular matrix $L = [\ell_{ij}]$ and its transpose,

$$A = LL^T,$$

where the diagonal elements of L are all positive.

Cholesky Factorization Algorithm

INPUT: The entries a_{ij}, $1 \le i, j \le n$, of the symmetric positive definite matrix A.

OUTPUT: The entries ℓ_{ij}, $1 \le j \le i \le n$, of the lower–triangular matrix L satisfying $A = LL^T$.

Step 1. For $j = 1, 2, \ldots, n$, compute

$$\ell_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} \ell_{jk}^2}.$$

If $a_{jj} - \sum_{k=1}^{j-1} \ell_{jk}^2 \le 0$, then OUTPUT (*Factorization impossible—A not positive definite*); STOP.

Step 2. For each $i = j+1, j+2, \ldots, n$, compute

$$\ell_{ij} = \frac{1}{\ell_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik}\ell_{jk} \right).$$

Step 3. Continue Steps 1 and 2 for $j = 1, 2, \ldots, n$ until all ℓ_{ij} are computed.

Step 4. OUTPUT (ℓ_{ij} for $1 \le j \le i \le n$); STOP.

Remarks:

$$A = LL^T, \qquad \ell_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} \ell_{jk}^2}, \qquad \ell_{ij} = \frac{1}{\ell_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik}\ell_{jk} \right).$$

1. The Cholesky factorization exists if and only if A is **symmetric positive definite**; that is, $A = A^T$ and $x^T Ax > 0$ for all nonzero vectors $x \in \mathbb{R}^n$.

2. Since $A = LL^T$, the system $A\mathbf{x} = \mathbf{b}$ can be solved efficiently by first solving

$$L\mathbf{y} = \mathbf{b}, \quad \text{then } L^T\mathbf{x} = \mathbf{y}.$$

3. The Cholesky method requires approximately $\frac{1}{3}n^3$ arithmetic operations—about one–half the work required for Gaussian elimination.

4. Pivoting is unnecessary because positive definiteness ensures stability and nonzero pivots.

Example 3:

$$A = \begin{bmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{bmatrix}.$$

Compute the elements of L:

$$\begin{aligned} \ell_{11} &= \sqrt{4} = 2, \\ \ell_{21} &= \frac{12}{2} = 6, \quad \ell_{31} = \frac{-16}{2} = -8, \\ \ell_{22} &= \sqrt{37 - 6^2} = 1, \\ \ell_{32} &= \frac{-43 - (-8)(6)}{1} = 5, \\ \ell_{33} &= \sqrt{98 - (-8)^2 - 5^2} = 3. \end{aligned}$$

Hence,

$$L = \begin{bmatrix} 2 & 0 & 0 \\ 6 & 1 & 0 \\ -8 & 5 & 3 \end{bmatrix}, \qquad L^T = \begin{bmatrix} 2 & 6 & -8 \\ 0 & 1 & 5 \\ 0 & 0 & 3 \end{bmatrix}.$$

Then $A = LL^T$, confirming the correctness of the factorization.

4.5 Iterative methods: Jacobi

The Jacobi method is an iterative technique for solving a system of n linear equations

$$A\mathbf{x} = \mathbf{b},$$

where $A = [a_{ij}]$ is an $n \times n$ nonsingular matrix and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. It is especially useful for large, sparse systems for which direct methods (such as Gaussian elimination or LU factorization) are computationally expensive.

The idea comes from splitting A into an diagonal matrix and an off-diagonal matrix as follows

$$A = D + R$$

where D is the diagonal matrix containing only the diagonal elements of A,

$$D = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix},$$

and a matrix R contains all the off-diagonal elements:

$$R = A - D = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ a_{21} & 0 & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & 0 \end{bmatrix}.$$

Next, substituting $A = D + R$ into the original system $A\mathbf{x} = \mathbf{b}$, we have:

$$(D + R)\mathbf{x} = \mathbf{b}.$$

Rearrange to isolate the diagonal termwe derive:

$$D\mathbf{x} = \mathbf{b} - R\mathbf{x}.$$

Notice that, since D is diagonal, it's easy to invert:

$$\mathbf{x} = D^{-1}(\mathbf{b} - R\mathbf{x})$$

In the Jacobi method, we use the previous iteration's vector $\mathbf{x}^{(k)}$ to compute the next one:

$$\mathbf{x}^{(k+1)} = D^{-1}\left(\mathbf{b} - R\mathbf{x}^{(k)}\right).$$

To derive the corresponding algorithm, we can rephrase the method as solving each equation for its diagonal variables:

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j\right), \quad i = 1, 2, \dots, n.$$

In the Jacobi iteration, each new value $x_i^{(k+1)}$ is computed using only values from the *previous iteration*:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, 2, \dots, n.$$

Algorithm of Jacobi Iterative Method

To solve the linear system $A\mathbf{x} = \mathbf{b}$ iteratively for $\mathbf{x}$.

INPUT: The coefficients a_{ij} and right-hand side terms b_i, $1 \leq i, j \leq n$; an initial approximation $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})^T$; a tolerance $TOL > 0$; and a maximum number of iterations N.

OUTPUT: An approximate solution $\mathbf{x}^{(k)}$ to $A\mathbf{x} = \mathbf{b}$ satisfying

$$\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_\infty < TOL.$$

Step 1. Set $k = 1$.

Step 2. For each $i = 1, 2, \ldots, n$, compute

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

Step 3. If $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_\infty < TOL$, then output $(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)})^T$; STOP.

Step 4. Set $k = k + 1$. If $k > N$, then the method failed to converge within N iterations. STOP.

Step 5. Return to Step 2.

Convergence Criterion:

The Jacobi method converges if the matrix A is strictly diagonally dominant:

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

or if A is symmetric positive definite.

Example 1: Use the Jacobi method to solve

$$\begin{cases} 10x_1 - x_2 + 2x_3 = 6, \\ -x_1 + 11x_2 - x_3 + 3x_4 = 25, \\ 2x_1 - x_2 + 10x_3 - x_4 = -11, \\ 3x_2 - x_3 + 8x_4 = 15, \end{cases}$$

with $\mathbf{x}^{(0)} = (0, 0, 0, 0)^T$.

$$\begin{aligned} x_1^{(k+1)} &= \tfrac{1}{10}(6 + x_2^{(k)} - 2x_3^{(k)}), \\ x_2^{(k+1)} &= \tfrac{1}{11}(25 + x_1^{(k)} + x_3^{(k)} - 3x_4^{(k)}), \\ x_3^{(k+1)} &= \tfrac{1}{10}(-11 - 2x_1^{(k)} + x_2^{(k)} + x_4^{(k)}), \\ x_4^{(k+1)} &= \tfrac{1}{8}(15 - 3x_2^{(k)} + x_3^{(k)}). \end{aligned}$$

After several iterations:

$$\begin{aligned} \mathbf{x}^{(1)} &= (0.6,\ 2.27,\ -1.1,\ 1.875)^T, \\ \mathbf{x}^{(2)} &= (1.09,\ 2.91,\ -0.97,\ 1.72)^T, \\ \mathbf{x}^{(3)} &= (1.02,\ 2.98,\ -1.00,\ 1.71)^T. \end{aligned}$$

Thus the solution converges to:

$$\mathbf{x} = (1.00,\ 3.00,\ -1.00,\ 1.70)^T.$$

Remarks:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

1. The Jacobi method is simple to implement but converges slowly compared with Gauss–Seidel or SOR methods.

2. For large sparse systems, it is easily parallelized because all $x_i^{(k+1)}$ are updated independently.

3. Converges if A is strictly diagonally dominant or symmetric positive definite. Convergence can fail if A is not diagonally dominant, but permuting the equations may help.

4.6 Iterative methods: Gauss-Seidel

The Gauss-Seidel method is an iterative refinement of the Jacobi method for solving the system of n linear equations

$$A\mathbf{x} = \mathbf{b},$$

where $A = [a_{ij}]$ is an $n \times n$ nonsingular matrix and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. Unlike the Jacobi method, the Gauss-Seidel method makes use of the most recently computed components of $\mathbf{x}$ as soon as they are available, thereby accelerating convergence.

Consider the system

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, \tag{3}$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \tag{4}$$
$$\ldots, \tag{5}$$
$$a_{n1}x_1 + \cdots + a_{nn}x_n = b_n. \tag{6}$$

Solving for the i th variable gives

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right), \quad i = 1, 2, \ldots, n.$$

In the Gauss-Seidel method, the most recent approximations are used immediately, which is the main difference from Jacobi method.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

Another interpretation of this method:
We decompose A into:

$$A = D + L + U$$

where $D =$ is the diagonal part of A, $L =$ is the lower triangular part, and $U =$ is the upper triangular part. Then, the Gauss-Seidel method uses the formula:

$$(D + L)x^{(k+1)} = -Ux^{(k)} + b.$$

Rerange it, we have:

$$x^{(k+1)} = D^{-1}\left(b - Ux^{(k)} - Lx^{k+1}\right).$$

Algorithm: Gauss-Seidel Iterative Method to solve the system $A\mathbf{x} = \mathbf{b}$ iteratively for x.

INPUT: The coefficients a_{ij} and right-hand side terms b_i, $1 \leq i, j \leq n$; an initial guess $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)})^T$; a tolerance $TOL > 0$, and a maximum number of iterations N.

OUTPUT: An approximate solution $\mathbf{x}^{(k)}$ such that

$$\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_\infty < TOL$$

Step 1. Set $k = 1$.

Step 2. For each $i = 1, 2, \ldots, n$, compute

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

Step 3. If $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_\infty < TOL$, then output $(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)})^T$; STOP.

Step 4. Set $k = k + 1$. If $k > N$, then output that the method failed to converge within N iterations. STOP.

Step 5. Return to Step 2.

Convergence Criterion.

The Gauss-Seidel method converges if A is:

1. Strictly diagonally dominant:

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

2. or Symmetric positive definite.

Example 1: Use the Gauss-Seidel method to solve

$$\begin{cases} 10x_1 - x_2 + 2x_3 = 6, \\ -x_1 + 11x_2 - x_3 + 3x_4 = 25, \\ 2x_1 - x_2 + 10x_3 - x_4 = -11, \\ 3x_2 - x_3 + 8x_4 = 15, \end{cases}$$

with $\mathbf{x}^{(0)} = (0, 0, 0, 0)^T$.

$$\begin{aligned} x_1^{(k+1)} &= \tfrac{1}{10}(6 + x_2^{(k)} - 2x_3^{(k)}), \\ x_2^{(k+1)} &= \tfrac{1}{11}(25 + x_1^{(k+1)} + x_3^{(k)} - 3x_4^{(k)}), \\ x_3^{(k+1)} &= \tfrac{1}{10}(-11 - 2x_1^{(k+1)} + x_2^{(k+1)} + x_4^{(k)}), \\ x_4^{(k+1)} &= \tfrac{1}{8}(15 - 3x_2^{(k+1)} + x_3^{(k+1)}). \end{aligned}$$

Blow lists a few results after few iterations:

$$\begin{aligned} \mathbf{x}^{(1)} &= (0.6,\ 2.33,\ -1.09,\ 1.80)^T, \\ \mathbf{x}^{(2)} &= (1.04,\ 2.99,\ -1.02,\ 1.72)^T, \\ \mathbf{x}^{(3)} &= (1.00,\ 3.00,\ -1.00,\ 1.70)^T. \end{aligned}$$

The solution converges to:

$$\mathbf{x} = (1.00,\ 3.00,\ -1.00,\ 1.70)^T.$$

Remarks:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

1. The Gauss-Seidel method typically converges faster than the Jacobi method because it uses the most recent approximations immediately.

2. Each iteration depends sequentially on the previous components, so the method is not easily parallelized.

3. Converges if A is strictly diagonally dominant or symmetric positive definite.

4.7 SOR

Gauss-Seidel treats the lower part as "new" and the upper part as "old," giving:

$$(D + L)x^{(k+1)} = b - Mx^{(k)}.$$

Define the Gauss-Seidel update $\hat{x}^{(k+1)}$ by:

$$(D + L)\hat{x}^{(k+1)} = b - Mx^{(k)}.$$

SOR (Successive Over-Relaxation) is a relaxed version of Gauss-Seidel: instead of going fully to $\hat{x}^{(k+1)}$, we move partway toward it using a relaxation factor ω :

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

Substituting $\hat{x}^{(k+1)} = (D+L)^{-1}\left(b - Mx^{(k)}\right)$ yields:

$$x^{(k+1)} = x^{(k)} + \omega \left[(D+L)^{-1}\left(b - Mx^{(k)}\right) - x^{(k)}\right]$$

or equivalently,

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

This is a valid SOR form, but it contains $(D+L)^{-1}$. To eliminate the inverse, multiply both sides by $D+L$:

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

Expanding and grouping terms gives:

$$(D+L)x^{(k+1)} = \omega b + [(1-\omega)(D+L) - \omega M]x^{(k)} = \omega b + [(1-\omega)D + (1-\omega)L - \omega M]x^{(k)}.$$

To match the standard form with $D+\omega L$ on the left, we move the $(1-\omega)Lx^{(k)}$ term to the left side, which produces the final SOR iteration:

$$(D+\omega L)x^{(k+1)} = \omega b - [\omega M + (\omega - 1)D]x^{(k)}.$$

4.8 Krylov methods: Conjugate Gradient Method and Preconditioning, Page 479

The Conjugate Gradient (CG) method is an efficient iterative technique for solving systems of linear equations

$$A\mathbf{x} = \mathbf{b},$$

where A is a symmetric positive definite matrix, (i.e., $A^T = A$ and $x^T Ax > 0$ for all $x \neq 0$).

Unlike Jacobi or Gauss-Seidel, which are relaxation methods, the Conjugate Gradient method is based on minimizing a quadratic function associated with the system. The motivation is this method lies in the following thinking. For symmetric positive definite A, the system $A\mathbf{x} = \mathbf{b}$ is equivalent to minimizing the quadratic functional:

$$f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}.$$

Since the gradient of $f(x)$ is:

$$\nabla f(x) = Ax - b$$

So minimizing $f(x)$ means finding where this gradient $\nabla f(x) = 0$, namely the equilibrium point. The solution $\hat{\mathbf{x}}$ satisfies $\nabla f(\hat{\mathbf{x}}) = A\hat{\mathbf{x}} - \mathbf{b} = 0$. The Conjugate Gradient method constructs a sequence $\{\mathbf{x}^{(k)}\}$ that converges to $\hat{\mathbf{x}}$ by choosing search directions that are the steepest descent, which would move along the negative gradient direction at each step:

$$x_{k+1} = x_k + \alpha_k r_k,$$

where $r_k = b - Ax_k$ is the residual (negative gradient) and α_k is the step size chosen to minimize f along that direction.

For instance, let $\mathbf{x}^{(0)}$ be the initial guess. Define the initial residual:

$$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)},$$

which is also the negative gradient of $f(\mathbf{x})$ at $\mathbf{x}^{(0)}$. The first search direction is:

$$\mathbf{p}^{(0)} = \mathbf{r}^{(0)}.$$

At each iteration, we move along $\mathbf{p}^{(k)}$ by a step length α_k that minimizes $f(\mathbf{x})$ along that direction:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)},$$

where the step size α_k is obtained by:

$$\alpha_k = \frac{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T A \mathbf{p}^{(k)}}.$$

Then we compute the new residual:

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A \mathbf{p}^{(k)}.$$

The new search direction $\mathbf{p}^{(k+1)}$ is chosen to be A-orthogonal to all previous directions:

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta_k \mathbf{p}^{(k)}, \quad \text{where} \quad \beta_k = \frac{(\mathbf{r}^{(k+1)})^T \mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}.$$

Repeat above steps until convergence.

Why α_k is chosen that way?

At iteration k, we have:

$$x_{k+1} = x_k + \alpha_k p_k.$$

We want to choose α_k so that x_{k+1} minimizes the quadratic energy:

$$f(x) = \frac{1}{2} x^T A x - b^T x.$$

Assume at iteration k, we have $x = x_k + \alpha p_k$, then:

$$\begin{aligned}
\phi(\alpha) &= f\left(x_k + \alpha p_k\right) \\
&= \frac{1}{2}\left(x_k + \alpha p_k\right)^T A \left(x_k + \alpha p_k\right) - b^T \left(x_k + \alpha p_k\right) \\
&= f\left(x_k\right) + \alpha \left(p_k^T A x_k - p_k^T b\right) + \frac{1}{2}\alpha^2 p_k^T A p_k.
\end{aligned}$$

To minimize $\phi(\alpha)$, set derivative to zero:

$$\frac{d\phi}{d\alpha} = p_k^T \left(A x_k - b\right) + \alpha p_k^T A p_k = 0.$$

By definition,

$$r_k = b - Ax_k \quad \Rightarrow \quad Ax_k - b = -r_k$$

So,

$$-p_k^T r_k + \alpha_k p_k^T A p_k = 0.$$

Therefore. we derive:

$$\alpha_k = \frac{p_k^T r_k}{p_k^T A p_k}.$$

Algorithm: Conjugate Gradient Method

INPUT: Matrix A, vector $\mathbf{b}$, initial guess $\mathbf{x}^{(0)}$, tolerance $TOL > 0$, and maximum number of iterations N.

OUTPUT: Approximate solution $\mathbf{x}^{(k)}$ such that

$$\|\mathbf{r}^{(k)}\|_2 < TOL.$$

Step 1. Compute $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$, set $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$, and $k = 0$.

Step 2. While $k < N$ and $\|\mathbf{r}^{(k)}\|_2 \geq TOL$, do:

$$\alpha_k = \frac{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T A \mathbf{p}^{(k)}},$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)},$$
$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A \mathbf{p}^{(k)}.$$

If $\|\mathbf{r}^{(k+1)}\|_2 < TOL$, then OUTPUT $\mathbf{x}^{(k+1)}$; STOP.

$$\beta_k = \frac{(\mathbf{r}^{(k+1)})^T \mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}},$$

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta_k \mathbf{p}^{(k)}.$$

Set $k = k + 1$ and repeat Step 2.

Remarks:

1. The method terminates in at most n iterations (exact arithmetic).
2. Each iterate $\mathbf{x}^{(k)}$ minimizes $f(\mathbf{x})$ over the Krylov subspace

$$\mathcal{K}_k = \text{span}\{\mathbf{r}^{(0)}, A\mathbf{r}^{(0)}, A^2\mathbf{r}^{(0)}, \ldots, A^{k-1}\mathbf{r}^{(0)}\}.$$

3. The residuals $\mathbf{r}^{(k)}$ are mutually orthogonal, and the search directions $\mathbf{p}^{(k)}$ are A-conjugate:

$$(\mathbf{r}^{(i)})^T \mathbf{r}^{(j)} = 0, \quad (\mathbf{p}^{(i)})^T A \mathbf{p}^{(j)} = 0 \quad (i \neq j).$$

4. The convergence rate depends on the condition number $\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$:

$$\frac{\|\mathbf{x}^{(k)} - \mathbf{x}^*\|_A}{\|\mathbf{x}^{(0)} - \mathbf{x}^*\|_A} \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k.$$

Example 4:

Solve

$$A\mathbf{x} = \mathbf{b}, \quad A = \begin{bmatrix} 4 & 1 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{x}^{(0)} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

$$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)} = \begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}9\\5\end{bmatrix} = \begin{bmatrix}-8\\-3\end{bmatrix}, \quad \mathbf{p}^{(0)} = \mathbf{r}^{(0)}.$$

$$\alpha_0 = \frac{(-8)^2 + (-3)^2}{(-8,-3)\begin{bmatrix}4 & 1\\1 & 3\end{bmatrix}\begin{bmatrix}-8\\-3\end{bmatrix}} = \frac{73}{331} \approx 0.22.$$

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \alpha_0\mathbf{p}^{(0)} = \begin{bmatrix}2\\1\end{bmatrix} + 0.22\begin{bmatrix}-8\\-3\end{bmatrix} = \begin{bmatrix}0.24\\0.34\end{bmatrix}.$$

Continuing this process yields the exact solution

$$\boxed{\mathbf{x}^* = (0.0909,\, 0.6364)^T.}$$

Remarks:

$$\begin{aligned}
\mathbf{r}^{(k)} &= \mathbf{b} - A\mathbf{x}^{(k)},\\
\alpha_k &= \frac{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T A\mathbf{p}^{(k)}},\\
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)},\\
\mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} - \alpha_k A\mathbf{p}^{(k)},\\
\beta_k &= \frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}},\\
\mathbf{p}^{(k+1)} &= \mathbf{r}^{(k+1)} + \beta_k\mathbf{p}^{(k)}.
\end{aligned}$$

1. The Conjugate Gradient method is suitable only for symmetric positive definite matrices.

2. It typically converges much faster than Jacobi or Gauss–Seidel, especially when A is well-conditioned.

3. In practice, preconditioning is often used to reduce the condition number of A and accelerate convergence.

4. Each iteration requires one matrix–vector product, two inner products, and several vector updates.