CSC336 Notes

Base	Integer Representation	Example
10	$x = (x_n \cdots x_0)_{10} = \sum_{i=0}^{n} 10^i x_i$	$350 = (350)_{10}$ $= 100(3) + 10(5) + 1(0)$
2	$x = (x_n \cdots x_0)_2 = \sum_{i=0}^{n} 2^i x_i$	$350 = (101011110)_2$ $= 256(1) + 128(0) + 64(1) + 32(0) + 16(1) + 8(1) + 4(1) + 2(1) + 1(0)$
b	$x = (x_n \cdots x_0)_b = \sum_{i=0}^{n} b^i x_i$	For $b > 10$, we use letters A, B, C, D, E, ... to represent 11, 12, 13, 14, 15, ...

- Computers operate internally in binary, but user-wise, display numbers in decimal for our convenience
- Computers distinguish positive/negative with an extra digit at the number's front
- Base 2 can simulate base 2^i, eg. $(101011110)_2 = \left(1\underbrace{01}_{1}\underbrace{01}_{1}\underbrace{11}_{3}\underbrace{10}_{2}\right)_2 = (11132)_4 = \left(1\underbrace{11}_{5}\underbrace{32}_{14}\right)_4 = (15\text{E})_{16}$

Base	Real Number Representation	Example
10	$x = \pm(x_n \cdots x_0 . x_{-1} \cdots)_{10}$ $= \pm\left(\sum_{i=0}^{n} 10^i x_i + \sum_{i=1}^{\infty} 10^{-i} x_{-i}\right)$	$0.\overline{6} = (0.\overline{6})_{10}$ $= 1(0) + 0.1(6) + 0.01(6) + 0.001(6) + 0.0001(6) + \cdots$
2	$x = \pm(x_n \cdots x_0 . x_{-1} \cdots)_2$ $= \pm\left(\sum_{i=0}^{n} 2^i x_i + \sum_{i=1}^{\infty} 2^{-i} x_{-i}\right)$	$0.1 = (0.0\overline{0011})_2$ $= 1(0) + 0.5(0) + 0.25(0) + 0.125(0) + 0.0625(1) + \cdots$
b	$x = \pm(x_n \cdots x_0 . x_{-1} \cdots)_b$ $= \pm\left(\sum_{i=0}^{n} b^i x_i + \sum_{i=1}^{\infty} b^{-i} x_{-i}\right)$	The ∞ in the summation isn't necessarily true if the decimal ends.

Integral Part ($x_I = x_n \cdots x_0$): The non-decimal part of a real number

Fraction ($x_F = x_{-1} \cdots$): The decimal part of a real number

- If $(x_F)_2$ terminates in n digits, then $(x_F)_{10}$ terminates in n digits, but not vice versa

Algorithm: Integers, Base 10 to Base b

```
def base_10_to_b(n, b):
    quotient = n
    result = []
    while quotient > 0:
        quotient, remainder = divmod(quotient, b)
        result.insert(b, 0)
    return result
```

quotient	b	remainder	result
350	2	0	0
175	2	1	10
87	2	1	110
43	2	1	1110
21	2	1	11110
10	2	0	011110
5	2	1	1011110
2	2	0	01011110
1	2	1	101011110

Algorithm: Fraction, Base 10 to Base b

```
def base_10_to_b(n, b):
    fraction = n
    result = []
    while product < 1:
        fraction, integral = divmod(fraction * b, 1)
        result.append(b)
    return result
```

fraction	b	fraction * b	integral	result
0.15	2	0.3	0	0.0
0.3	2	0.6	0	0.00
0.6	2	1.2	1	0.001
0.2	2	0.4	0	0.0010
0.4	2	0.8	0	0.00100
0.8	2	1.6	1	0.001001
0.6	2	1.2	1	0.0010011
...	2	...	...	$0.00\overline{1001}$

Floating-Point Number: Computer representations of real numbers, usually in binary. Has fixed precision. Takes inspiration from scientific notation (eg. $\pm 0.0510 \times 10^{200}$)

- **Exponent/Characteristic (e):** An integer containing all "exponent information" (eg. 10^{200})
- **Mantissa/Significand (f):** A fraction containing all "digit information", no sign information (eg. 0.0510)
  - **Normalized:** A floating-point number, if the mantissa is all zeros or its 1st digit is nonzero. (eg. 0.0510×10^{200} is not normalized, 5.10×10^{198} is normalized)
  - **Significant Digits:** All digits after the mantissa's 1st nonzero digit (eg. 510)
- **Overflow Level (OFL/$n_{\max}$):** The largest-magnitude floating-point number
  - **Overflow:** When floating-point x, where $|x| > |n_{\max}|$, is stored on the computer
- **Underflow Level (UFL/$n_{\min}$):** The smallest-magnitude floating-point number
  - **Underflow:** When floating-point x, where $|x| < |n_{\min}|$, is stored on the computer

$x = (f)_b \times b^{(e)_b}$ $= \pm(.\, f_1 \cdots f_t)_b \times b^{\pm(e_{s-1}\cdots e_0)_b}$	Simplified Model *t-digit precision, base b*

- $-1 < (f)_b < 1$
- Let $a = b - 1$, then
  - $e_{\max} = -e_{\min} = (a \cdots a)_b$
  - $n_{\max} = (.\, a \cdots a)_b \times b^{(a\cdots a)_b}$
  - $n_{\min} = \begin{cases} (.\, 10\cdots 0)_b \times b^{-(a\cdots a)_b} & \text{if normalized (assume this by default)} \\ (.\, 0\cdots 01)_b \times b^{-(a\cdots a)_b} & \text{if not normalized} \end{cases}$

- $\mathbb{R}_b(t,s)$ is the set of base b floating-point numbers representable with a t-digit mantissa (including sign) and s-digit exponent (including sign)
- $\mathbb{R}_b(t,s) \subseteq [-n_{\max}, -n_{\min}] \cup \{0\} \cup [n_{\min}, n_{\max}]$
- $\mathbb{R}_b(t,s)$ is finite and not dense, while $\mathbb{R}$ is infinite and dense
- $\mathbb{R}_b(t,s)$ is mostly concentrated around 0
- $\mathbb{R}_b(t,s)$ is not closed with respect to mathematical operations (ie. $x * y \in \mathbb{R}_b(t,s)$ not guaranteed)
- To convert $\mathbb{R}$ to $\mathbb{R}_b(t,s)$, that is apply fl: $\mathbb{R} \to \mathbb{R}_b(t,s)$*, normalize the mantissa and:

Chopping	Traditional Rounding	Proper/Perfect Rounding
Ignore everything after the t-th digit in the mantissa	Round d_t up if $d_{t+1} \geq \frac{b}{2}$ and down otherwise, then chop.	Round d_t up if $d_{t+1} > \frac{b}{2}$, down if $d_{t+1} < \frac{b}{2}$, and to the nearest even d_t value if $d_{t+1} = \frac{b}{2}$.
eg. 2 digits, base 10 $-0.305 \to -0.30$ $-0.315 \to -0.31$ $-0.3155 \to -0.31$ $-0.3055 \to -0.30$	eg. 2 digits, base 10 $-0.305 \to -0.31$ $-0.315 \to -0.32$ $-0.3155 \to -0.32$ $-0.3055 \to -0.31$	eg. 2 digits, base 10 $-0.305 \to -0.30$ $-0.315 \to -0.32$ $-0.3155 \to -0.32$ $-0.3055 \to -0.31$
eg. 2 digits, base 2 $0.101 \to 0.10$ $0.111 \to 0.11$	eg. 2 digits, base 2 $0.101 \to 0.11$ $0.111 \to 1.0$	eg. 2 digits, base 2 $0.101 \to 0.10$ $0.111 \to 1.0$

**Assume no overflow/underflow*

Round-Off Error: The difference between $x - \text{fl}(x)$, roughly proportional to x

- **Relative Round-Off Error (δ):** Proportional round-off error, derived from $\text{fl}(x) = x(1+\delta)$
- **Unit Round-Off (μ/u):** A bound around δ, equal to $\mu = \begin{cases} b^{1-t} & \text{if normalized, chopping} \\ \frac{1}{2}b^{1-t} & \text{if normalized, rounding} \end{cases}$

Saturation: Phenomenon where $\text{fl}(x+y) = \text{fl}(x)$ where $y \neq 0$. Occurs when $|x| \gg |y|$.

Machine Epsilon: The smallest non-normalized floating-point ϵ_{mach} such that $\text{fl}(\epsilon_{\text{mach}} + 1) > 1$, equal to...

Chopping	Traditional Rounding	Proper/Perfect Rounding
$\epsilon_{\text{mach}} = b^{1-t}$	$\epsilon_{\text{mach}} = \frac{1}{2}b^{1-t}$	$\epsilon_{\text{mach}} = \frac{1}{2}b^{1-t} + b^{-t}$

- $0 < n_{\min} < \delta \leq \epsilon_{\text{mach}} < n_{\max}$

Correct in r Significant b-Digits: An approximation $\hat{x}$ to x, if $\left|\frac{x-\hat{x}}{x}\right| \leq \frac{1}{2}b^{1-r}$

Absolute Error: Of an approximation $\hat{x}$ to x, the value $x - \hat{x}$

Relative Error: Of an approximation $\hat{x}$ to x, the value $\frac{x-\hat{x}}{x}$

$x = (f)_2 \times b^{(e)_2}$
$= (-1)^q \times (f_0 . f_1 \cdots f_{t-1})_2 \times 2^{(-1)^p \times (e_{s-2} \cdots e_0)_2}$

Institute of Electrical & Electronics Engineers (IEEE) Standard

Simplified Version, Base 2

- $p, q, f_i, e_i \in \{0,1\}$
- $e_{\max} = -e_{\min} + 1 = (1 \cdots 1)_2$
- $n_{\max} = (1.1 \cdots 1)_2 \times 2^{(1 \cdots 1)_1}$
- $n_{\min} = (1.00 \cdots 0)_2 \times 2^{-(1 \cdots 1)_1}$

- Used by most modern computer systems
- Includes "special" numbers like $\pm\infty$ and NaN (not a number) for indeterminate values
- Uses proper rounding
- Needs $t + s$ bits to store $q, f_1 \ldots, f_{t-1}, p, e_0, \ldots, e_{s-2}$
  - As the mantissa is normalized ($f_0 \neq 0$) and IEEE is base 2, then $f_0 = 1$, so we don't store f_0.

Precision	Bits	$(t)_2$	$(s)_2$	$(e_{\min})_2$	$(e_{\max})_2$	$(t)_{10}$	$(e_{\min})_{10}$	$(e_{\max})_{10}$	ϵ_{mach}
Single (binary 32)	32	24	8	-126	127	7	-38	38	2^{-23}
Double (binary64)	64	53	11	-1022	1023	16	-308	308	2^{-52}
Quadruple (binary128)	128	113	15	-16382	16383	34	-4928	4928	2^{-112}

Computer Arithmetic

For $x, y \in \mathbb{R}_b(t,s)$, function $* \in \{+,-,\times,/\}$, and its floating-point equivalent $\overline{*}$, we usually have

$$x * y \neq \text{fl}(x \,\overline{*}\, y)$$
$$x \,\overline{*}\, y = \text{fl}(x * y) = (x * y)(1+\delta)$$
$$\text{fl}(x) = x(1+\delta)$$

- Behaviour varies depending on the machine, but is usually true for built-in functions (eg. $\sin x$)
- Often, computer uses 1-2 temporary digits
- Create a different δ for each operation, and bound them using $-\epsilon_{\text{mach}} \leq \delta \leq \epsilon_{\text{mach}}$.

eg. Multiplication

$$\begin{aligned}\text{fl}(x) \,\overline{\times}\, \text{fl}(y) &= \text{fl}\big(\text{fl}(x) \times \text{fl}(y)\big) \\ &= \text{fl}\big(x(1+\delta_1)y(1+\delta_2)\big) \\ &= x(1+\delta_1)y(1+\delta_2)(1+\delta_3) \\ &\approx xy[1+(\delta_1+\delta_2+\delta_3)]\end{aligned}$$

Define $\delta_\times = \delta_1 + \delta_2 + \delta_3$

$$|\delta_\times| \leq 3\epsilon_{\text{mach}}$$

We can ignore δ or ϵ_{mach} when they are multiplied with each other in our math, because they are already extremely tiny; multiplying them makes them negligible.

eg. Addition

$$\begin{aligned}\text{fl}(x) \,\overline{+}\, \text{fl}(y) &= \text{fl}\big(\text{fl}(x) + \text{fl}(y)\big) \\ &= \text{fl}\big(x(1+\delta_1) + y(1+\delta_2)\big) \\ &= [x(1+\delta_1) + y(1+\delta_2)](1+\delta_3) \\ &\approx x(1+\delta_1+\delta_3) + y(1+\delta_2+\delta_3) \\ &= (x+y) + x(\delta_1+\delta_3) + y(\delta_2+\delta_3) \\ &= (x+y)\left(1 + \frac{x(\delta_1+\delta_3)}{x+y} + \frac{y(\delta_2+\delta_3)}{x+y}\right)\end{aligned}$$

Define $\delta_+ = \frac{x(\delta_1+\delta_3)}{x+y} + \frac{y(\delta_2+\delta_3)}{x+y}$

$$\begin{aligned}|\delta_+| &\leq \left|\frac{x}{x+y}\right| \cdot 2\epsilon_{\text{mach}} + \left|\frac{y}{x+y}\right| \cdot 2\epsilon_{\text{mach}} \\ &= 2\epsilon_{\text{mach}} \frac{|x|+|y|}{|x+y|} \\ &= \begin{cases} 2\epsilon_{\text{mach}} & xy > 0 \\ 2\epsilon_{\text{mach}} \dfrac{|x-y|}{|x+y|} & xy < 0 \end{cases}\end{aligned}$$

For $x+y$ with $x \approx -y$, we have $\frac{|x-y|}{|x+y|} \to \infty$, resulting in huge error called **catastrophic cancellation**.

Relative Condition Number: Sensitivity of $f(x)$ to small changes in input; how strongly it propagates relative error

$$\kappa_f(x) = \left|x \frac{f'(x)}{f(x)}\right|$$

Well-Conditioned: A function, if κ_f is small; small changes in input produce small changes on output

Ill-Conditioned: A function, if κ_f is large; small changes in input produce large changes on output

- Use $\kappa_f = \frac{1}{\epsilon_{\text{mach}}}$ as general threshold for well-conditioning – above that, unlikely that any digits are correct

eg. $f(x) = \sqrt{x}$

$$\kappa_f = \left|x \cdot \frac{\left(\frac{1}{2}x^{-\frac{1}{2}}\right)}{\sqrt{x}}\right| = \left|x \cdot \frac{1}{2x}\right| = \frac{1}{2}$$

κ_f is small and independent of x; well-conditioned

eg. $f(x) = e^x$

$$\kappa_f = \left|x \cdot \frac{e^x}{e^x}\right| = |x|$$

κ_f depends linearly on $|x|$, only overflowing for large $|x|$, so we say it is well-conditioned for all acceptable x values

Stability: Sensitivity of a numerical algorithm (how a computation is carried out) to small changes in input

eg. 3 significant digits, rounding, $f(x,y) = (x-y)^2 = x^2 - 2x + y^2$

For $x = 15.6, y = 15.7$,

$$\begin{aligned}(x-y)^2 &= (15.6-15.7)^2 \\ &= (-0.1)^2 \\ &= 0.01\end{aligned} \qquad \begin{aligned}x^2 - 2x + y^2 &= 243.36 - 2(244.92) + 246.49 \\ &\to 243 - 490 + 269 \\ &= -1 \text{ (very wrong!)}\end{aligned}$$

It is clear that $(x-y)^2$ is a more stable algorithm than $x^2 - 2x + y^2$.

- Mathematically-equivalent expressions are not always computationally equivalent
- Stabler algorithms for computations will not change condition number
- No general rule for the most stable algorithm, but some heuristics:
  - Avoid subtracting nearly-equal numbers (avoid catastrophic cancellation)
  - Minimize number of operations (avoid propagation of error)
  - Add numbers from smallest to largest (so the mantissa does not drop off digits too early on)
  - Be careful adding numbers of different scales

ADD EXAMPLE USING EPSILON MACHINE!!!!

Symbol	Meaning
x	Input, exact
$\hat{x}$	Input, approximate
$y = f(x)$	Output, exact
$\hat{y} = f(\hat{x}) = \hat{f}(x)$	Output, approximate – exact solution to modified input or modified solution to exact input
$g(x) \approx \hat{f}(x)$	Mathematical approximation of $f(x)$, exact
$\hat{g}(x)$	Mathematical approximation of $f(x)$, approximate

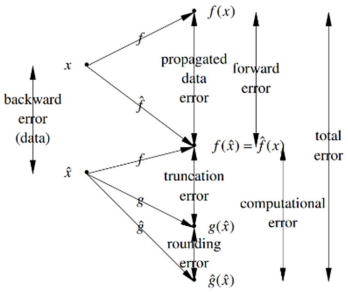
Forward Error: The difference $y - \hat{y}$

Backward Error: The difference $x - \hat{x}$, where $\hat{x} = f^{-1}(\hat{y}) = f^{-1}\left(\hat{f}(x)\right)$

Relative Forward Error: The value $\frac{y-\hat{y}}{y}$. Often easier to calculate.

Relative Backward Error: The value $\frac{x-\hat{x}}{x}$. Often harder to calculate.

$$\left|\frac{y-\hat{y}}{y}\right| \approx \kappa_f \times \left|\frac{x-\hat{x}}{x}\right|$$



Rounding Error: Error from propagated rounding of limited-precision numbers

Truncation/Discretization Error: Error from computing approximations of math expressions (eg. Taylor series)

Computational Error: Sum of rounding and truncation error

$$\underbrace{f(x) - \hat{g}(\hat{x})}_{\text{total error}} = \underbrace{f(x) - f(\hat{x})}_{\text{propagated data error}} + \underbrace{f(\hat{x}) - g(\hat{x})}_{\text{truncation error}} + \underbrace{g(\hat{x}) - \hat{g}(\hat{x})}_{\text{rounding error}}$$

Taylor's Theorem: For $f: \mathbb{R} \to \mathbb{R}$, there exists ξ between x and a,

$$\begin{aligned}f(x) &= t_k(x) + R_{k+1}(x) \\ &= \underbrace{f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k}_{k\text{th Taylor polynomial}} + \underbrace{\frac{f^{(k+1)}(\xi)}{(k+1)!}(x-a)^{k+1}}_{\text{Remainder}}\end{aligned}$$

- An equivalent form is $f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \cdots + \frac{f^{(k)}(x)}{k!}h^k + \frac{f^{(k+1)}(\xi)}{(k+1)!}h^{k+1}$
- When $x \approx a$, $R_k(x) \approx 0$, then $f(x) = t_k(x) + R_{k+1}(x) \approx t_k(x)$, so Taylor Series is a good approximation
- Truncation error is $R_k(x)$

eg. Compute total error of $\sin\frac{\pi}{8} = 0.38268343\ldots$ using 3-decimal digit floating-point arithmetic.

Approximate $x = \frac{\pi}{8}$ with $\hat{x} = \frac{3}{8}$.

Approximate $f(x) = \sin x$ with $g(x) = x - \frac{x^3}{3!}$

$$\hat{g}(\hat{x}) = \hat{g}\left(\frac{3}{8}\right) = \text{fl}\left[\text{fl}\left(\frac{3}{8}\right) - \text{fl}\left[\text{fl}\left(\frac{1}{3!}\right)\text{fl}\left[\text{fl}\left(\frac{3}{8}\right)^3\right]\right]\right] \approx 0.366$$

Backward Error	*Propagated Data Error*	*Truncation Error*
$x - \hat{x} = \pi - 3$ ≈ 0.14159	$f(x) - f(\hat{x}) = \sin\frac{\pi}{8} - \sin\frac{3}{8}$ ≈ 0.01641	$f(\hat{x}) - g(\hat{x}) = \sin\frac{3}{8} - \left(\frac{3}{8} - \frac{\left(\frac{3}{8}\right)^3}{3!}\right)$ $\approx 6.1591 \times 10^{-5}$

Rounding Error	*Total Error*
$g(\hat{x}) - \hat{g}(\hat{x}) = \left(\frac{3}{8} - \frac{\left(\frac{3}{8}\right)^3}{3!}\right) - \text{fl}\left[\text{fl}\left(\frac{3}{8}\right) - \text{fl}\left[\text{fl}\left(\frac{1}{3!}\right)\text{fl}\left[\text{fl}\left(\frac{3}{8}\right)^3\right]\right]\right]$ $= 2.109375 \times 10^{-4}$	$f(x) - \hat{g}(\hat{x}) = 0.38268343\ldots 0.366$ $= 0.01668343\ldots$

Linear Algebra

Real Matrix: A matrix $A \in \mathbb{R}^{m\times n}$

Order/Size: Of matrix $A \in \mathbb{C}^{m\times n}$, value $m \times n$

Symmetric Matrix: Square matrix A where $A = A^{\mathrm{T}}$

- If A is invertible, then A^{-1} is symmetric

Orthogonal Matrix: Real matrix A where $A^{\mathrm{T}}A = \mathbf{I}$.

- If A is square, then $A^{\mathrm{T}} = A^{-1}$

$(A^{-1})^{-1} = A$
$(AB)^{-1} = B^{-1}A^{-1}$
$(\alpha A)^{-1} = \frac{1}{\alpha}A^{-1}$
$\det AB = \det BA$
$= \det A \det B$

$(A^{\mathrm{T}})^{\mathrm{T}} = A$
$(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$
$(A^{\mathrm{H}})^{\mathrm{H}} = A$
$(AB)^{\mathrm{H}} = B^{\mathrm{H}}A^{\mathrm{H}}$
A is invertible
$\Leftrightarrow$ A^{T} is invertible

Determinant: Of square matrix A, the value $\det A = \begin{cases} A_{1,1} & \text{if } A \text{ is } 1\times 1 \\ \sum_{k=1}^{n}(-1)^{k+1}A_{1,k}\det(A'_{1,k}) & \text{else} \end{cases}$

where $A'_{i,j}$ is A but with row i, column j deleted

Square Root: Of matrix A, the matrix $A^{\frac{1}{2}}$ where $A = A^{\frac{1}{2}}A^{\frac{1}{2}}$

- If A is diagonal, $A^{\frac{1}{2}}$ is diagonal and $A^{\frac{1}{2}}_{i,i} = \sqrt{A_{i,i}}$

If A is square, then

A is not invertible/singular
$\Leftrightarrow$ $A\mathbf{x} = 0$ has non-trivial solution
$\Leftrightarrow$ $\det A = 0$
$\Leftrightarrow$ A columns are linearly dependent
$\Leftrightarrow$ Solutions to $A\mathbf{x} = \mathbf{b}$ are infinite if existing

A is invertible/non-singular
$\Leftrightarrow$ $A\mathbf{x} = 0$ has only the trivial solution
$\Leftrightarrow$ $\det A \neq 0$
$\Leftrightarrow$ A columns are linearly independent
$\Leftrightarrow$ Solutions to $A\mathbf{x} = \mathbf{b}$ are always unique

Complex Conjugate: Of complex number $z = a + bi \in \mathbb{C}$, the value $\overline{z} = a - bi \in \mathbb{C}$.

Conjugate Transpose: Of matrix $A \in \mathbb{C}^{m\times n}$, the $n \times m$ matrix $A^{\mathrm{H}} = \overline{A}^{\mathrm{T}}$ where $A^{H}_{i,j} = \overline{A_{j,i}}$.

Hermitian Matrix: Matrix A where $A^{\mathrm{H}} = A$.

- If A is real, then $A^{\mathrm{H}} = A^{\mathrm{T}}$

Unitary Matrix: Complex matrix A where $A^{\mathrm{H}}A = \mathbf{I}$.

- If A is square, then $A^{\mathrm{H}} = A^{-1}$

Normal Matrix: Matrix A where $A^{\mathrm{H}}A = AA^{\mathrm{H}}$

Density: Of matrix A, the value $\frac{\#\text{ of nonzero elements}}{\#\text{ of elements}}$ *("Dense" means most elements are non-zero)*

Sparsity: Of matrix A, the value $\frac{\#\text{ of zero elements}}{\#\text{ of elements}}$ *("Sparse" means most elements are zero)*

Permutation Matrix: The matrix P, equal to $\mathbf{I}$ but with some rows/columns swapped.

- Switching rows i, j of matrix A is equivalent to PA, where $P = \mathbf{I}$ with rows i, j swapped
- P is orthogonal
- $P_1 P_2$ is a permutation matrix

Elementary Permutation Matrix: A permutation matrix with only 2 rows/columns different from I.

- $P = P^{-1} = P^{\mathrm{T}}$
- $PP = \mathbf{I}$ *("idempotent")*

Inner Product: Of vectors $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$, the value $(\mathbf{u}, \mathbf{v}) = \mathbf{u}^{\mathrm{H}}\mathbf{v} \in \mathbb{C}$

Orthogonal: Vectors $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$, if $\mathbf{u}^{\mathrm{H}}\mathbf{v} = 0$.

- $\mathbf{u}$ and $\mathbf{v}$ are linearly independent

Orthonormal: Vectors $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$, if $\mathbf{u}^{\mathrm{H}}\mathbf{v} = 0$ and $\|\mathbf{u}\| = \|\mathbf{v}\| = 1$

Diagonal Matrix: Square matrix A where $A_{i,j} = 0$ for $i \neq j$

Lower Triangular Matrix: Square matrix A where $A_{i,j} = 0$ for $i < j$ *("Strictly lower triangular" is $i \leq j$)*

Upper Triangular Matrix: Square matrix A where $A_{i,j} = 0$ for $i > j$ *("Strictly upper triangular" is $i \geq j$)*

Unit Lower Triangular Matrix: Lower triangular matrix A where $A_{i,i} = 1$

Unit Upper Triangular Matrix: Upper triangular matrix A where $A_{i,i} = 1$

- If is A triangular or diagonal and $A_{i,i} \neq 0$, A is invertible
- Products and inverses of (unit) L/U triangular matrices are (unit) L/U triangular matrices

Diagonal: Of matrix A, the values $A_{i,i}$

l-th Subdiagonal: Of matrix A, the values $A_{i,i+l}$, where $l \in \mathbb{N}$ is the **lower bandwidth**

u-th Superdiagonal: Of matrix A, the values $A_{i,i-u}$, where $u \in \mathbb{N}$ is the **upper bandwidth**

Full/Total Bandwidth: The value $l + u + 1$

Banded Matrix: Sparse matrix A where all non-zero elements are near its diagonal.
(ie. values below l-th subdiagonal, above u-th superdiagonal are 0 for some l, u)

Symmetrically-Banded Matrix: Banded matrix A where $l = u$, where $l = u$ is the **semi-bandwidth**

- **Tridiagonal Matrix:** Symmetrically-banded matrix with semi-bandwidth 1
- **Pentadiagonal Matrix:** Symmetrically-banded matrix with semi-bandwidth 2

(Row) Diagonally-Dominant Matrix: Square matrix A where a diagonal's magnitude is at least equal to the sum of magnitudes of other values in its row

- Formally, $2|A_{i,i}| \geq \sum_{j=1}^{n} |A_{i,j}|$
- Strictly diagonally-dominant matrices (ie. $>$ instead of $\geq$) are invertible

Monotone: Square invertible matrix A where $A^{-1} \geq 0$

Positive: Matrix A where $A_{i,j} > 0$

Non-Negative: Matrix A where $A_{i,j} \geq 0$

Non-Positive: Matrix A where $A_{i,j} \leq 0$

Negative: Matrix A where $A_{i,j} < 0$

Positive Definite: Square matrix A where $\mathbf{x}^{\mathrm{T}} A \mathbf{x} > 0$ for $\mathbf{x} \neq \mathbf{0}$

Positive Semi-Definite: Square matrix A where $\mathbf{x}^{\mathrm{T}} A \mathbf{x} \geq 0$ for $\mathbf{x} \neq \mathbf{0}$

Negative Semi-Definite: Square matrix A where $\mathbf{x}^{\mathrm{T}} A \mathbf{x} \leq 0$ for $\mathbf{x} \neq \mathbf{0}$

Negative Definite: Square matrix A where $\mathbf{x}^{\mathrm{T}} A \mathbf{x} < 0$ for $\mathbf{x} \neq \mathbf{0}$

M-Matrix: A monotone, non-positive matrix $A \in \mathbb{R}^{m \times n}$

Rank: Of matrix A, the number of linearly independent columns/rows; dimension of its columns/rows' subspace
Null Space/Kernel: Of matrix A, set $N(A) = \{\mathbf{x} \in \mathbb{R}^n : A\mathbf{x} = 0\}$
Nullity: Dimension of null space

- If A is invertible, its nullity is 0

Range Space: Of matrix A, set $R(A) = \{\mathbf{v} \in \mathbb{R}^n : A\mathbf{x} = \mathbf{v} \text{ for some } \mathbf{x} \in \mathbb{R}^n\}$; the "image" of the matrix

- $A\mathbf{x} = \mathbf{b}$ has solution $\Leftrightarrow \mathbf{b} \in R(A)$
- If A is square, $\text{rank}(A) + \text{nullity}(A) = \text{order}(A)$

Term	Description
$\mathcal{O}(n^\beta) = \sum_{i=0}^{\beta} c_i n^i$	Asymptotic complexity of algorithms with problem size n, where $n \to \infty$
$\mathcal{O}(h^\alpha) = \sum_{\alpha=0}^{\infty} c_\alpha h^\alpha$	Asymptotic behaviour of error from a computational/discretization method, in terms of distance between "refinements" of the discretization, where $h \to 0$

- Note that $h = \frac{b-a}{n}$, so $\mathcal{O}(h^\alpha) = \mathcal{O}(n^{-\alpha})$

Gaussian Elimination

Forward Substitution: Method for solving for $\mathbf{x}$ in linear systems (ie. $A\mathbf{x} = \mathbf{b}$) where A is lower-triangular

Algorithm:

$$A\mathbf{x} = \mathbf{b}$$

$$\begin{bmatrix} A_{1,1} & 0 & 0 \\ \vdots & \ddots & 0 \\ A_{n,1} & \cdots & A_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

$$\begin{cases} A_{1,1}x_1 & & & & = b_1 \\ A_{2,1}x_1 + A_{2,2}x_2 & & & & = b_2 \\ \quad\vdots & & & & \\ A_{i,1}x_1 + & \ldots & + A_{i,i}x_i & & = b_i \\ \quad\vdots & & & & \\ A_{n,1}x_1 & + & \ldots & + A_{n,n} & = b_n \end{cases}$$

To solve each equation from top to bottom:
(1) Divide by $A_{1,1}$
(2) Subtract $A_{2,1}x_1$, divide by $A_{2,2}$, substitute x_1
(3) Subtract $A_{1,1}x_1 + A_{2,1}x_2$, divide by $A_{3,3}$, substitute x_1, x_2
(i) Subtract $\sum_{j=1}^{i-1} A_{j,i}$, divide by $A_{i,i}$, substitute $x_1, \ldots, x_i$

Therefore the general solution set is

$$\begin{cases} x_1 = \dfrac{b_1}{A_{1,1}} \\ x_2 = \dfrac{b_2 - A_{2,1}x_1}{A_{2,2}} \\ \quad\vdots \\ x_i = \dfrac{b_i - \sum_{j=1}^{i-1} A_{i,j}x_j}{A_{i,i}} \\ \quad\vdots \\ x_n = \dfrac{b_n - \sum_{j=1}^{n-1} A_{n,j}x_j}{A_{n,n}} \end{cases}$$

Note that if $A_{i,i} = 0$, the system is not linearly independent and x_i has infinitely many possible values.

```
# Pretend x, b are length-n arrays, A is a n x n matrix
for i in range(n):
    x[i] = b[i]
    for j in range(i):
        x[i] -= A[i, j] * x[j]  # Equivalent to b_i - sum_{j=1}^{i-1} A_{i,j} x_j
    if A[i, i] != 0:
        x[i] /= A[i, i]  # Divide by A_{i,i}
# Running-time is O(n^2); needs sum_{i=1}^{n} i ≈ n^2/2 multiplications and subtractions, ≈ n divisions
```

Backward Substitution: Method for solving $A\mathbf{x} = \mathbf{b}$ where A is upper-triangular. Exact same as forward substitution, except equations are solved bottom to top (ie. i iterates $n-1$ to 0) with $x_i = \frac{b_i - \sum_{j=i+1}^{n} A_{i,j}x_j}{A_{i,i}}$

Flop: A paired instance of an addition/subtraction and a multiplication.

Algorithm – $(l, 0)$-Banded Case (Forward):

When j iterates from 1 to $i-1$, if $j > l$, the first $i-l$ terms will be 0, so we can simplify the i-th step to

(i) Subtract $\sum_{j=\max\{1,i-l\}}^{i-1} A_{j,i}$, divide by $A_{i,i}$, substitute $x_1, \dots, x_i$

```
for j in range(max(0, i - l), i):
    x[i] -= A[i, j] * x[j]

# Requires ≈ ∑_{i=1}^{n} (l+1) ≈ nl flops
```

Algorithm – $(0, u)$-Banded Case (Backward):

Similar to above, we simplify the i-th step to

(i) Subtract $\sum_{j=i+1}^{\min\{n,i+u\}} A_{j,i}$, divide by $A_{i,i}$, substitute $x_{i+1}, \dots, x_n$

```
for j in range(i, min(n, i + u)):
    x[i] -= A[i, j] * x[j]

# Requires ≈ ∑_{i=1}^{n} (u+1) ≈ nu flops
```

Equivalent: Two linear systems with the same solutions; $A\mathbf{x} = \mathbf{b} \Leftrightarrow C\mathbf{x} = \mathbf{d}$. Equivalence is preserved under:

- Row multiplied by scalar $\quad \rho_i \leftarrow \alpha\rho_i$
- Row addition $\quad \rho_i \leftarrow \rho_i + \rho_j$
- Row permutation $\quad \rho_i \leftrightarrow \rho_j$

Gaussian Elimination (GE): Method for solving linear systems by turning them into equivalent triangular system

GE Algorithm:

$$A\mathbf{x} = \mathbf{b}$$

$$\begin{bmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & \ddots & \vdots \\ A_{n,1} & \cdots & A_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Choose $A_{1,1}$ as a **pivot**

To set $A_{2,1}, \dots, A_{n,1} = 0$, apply $\rho_j \leftarrow \rho_j - \frac{A_{j,1}}{A_{1,1}}\rho_1$ to all rows ρ_j:

This cancels out the row's 1st element:

$$\frac{A_{j,1}}{A_{1,1}}\rho_1 = \frac{A_{j,1}}{A_{1,1}}[A_{1,1} \quad A_{1,2} \quad \cdots \quad A_{1,n}|b_1]$$

$$= \left[A_{j,1} \quad \frac{A_{j,1}}{A_{1,1}}A_{1,2} \quad \cdots \quad \frac{A_{j,1}}{A_{1,1}}A_{1,n} \middle| \frac{A_{j,1}}{A_{1,1}}A_{1,2}b_1\right]$$

$$\rho_j = [A_{j,1} \quad A_{j,2} \quad \cdots \quad A_{j,n}|b_j]$$

Thus, $\rho_j - \frac{A_{j,1}}{A_{1,1}}\rho_1$ results in $A_{j,1} = 0$.

Repeat this process for column $i \in \{1, \dots, n\}$:

Choose $A_{i,i}$ as a **pivot**

To set $A_{i+1,i}, \dots, A_{n,i} = 0$, apply $\rho_j \leftarrow \rho_j - \frac{A_{j,i}}{A_{i,i}}\rho_1$:

Results in upper-triangular matrix; use backward substitution!

```
# Gaussian Elimination, Basic
# Some variations use different loop orders

for i in range(n - 1):
    for j in range(i + 1, n):
        # Store A_{j,i}/A_{i,i} directly in A_{j,i}, which would
        # be 0 anyways, to save space
        if A[i, i] != 0:
            A[j, i] /= A[i, i]
        else:
            # Uh oh! Need a different pivot!

        # Apply row reduction
        A[j, i + 1:] -= A[j, i] * A[i, i + 1:]
        # for k in range(i + 1, n):
        #     A[j, k] -= A[j, i] * A[i, k]
        b[j] -= A[j, i] * b[i]

# The running-time is O(n^3)
# Editing A is ≈ n^3/3 flops, ≈ n^2/2 divisions
# Editing b is ≈ n^2/2 flops
```

GE Algorithm – Symmetric Case

- $A_{j,k} \leftarrow A_{j,k} - \frac{A_{j,i}}{A_{i,i}}A_{j,i}$ is identical to $A_{k,j} \leftarrow A_{k,j} - \frac{A_{k,i}}{A_{i,i}}A_{i,j}$
- Step i of GE preserves symmetry of bottom-right $(n-i) \times (n-i)$ submatrix
- We can update columns during row operations, halving flops of A

```
for k in range(j, n):
    A[j, k] -= c * A[i, k]
    A[k, j] = A[j, k]
```

LU Decomposition/Factorization

LU Decomposition/Factorization: Given matrix A, the upper-triangular Gaussian elimination U, and the strictly-unit lower triangular matrix L with the lower triangle made of $\frac{A_{j,i}}{A_{i,i}}$ values, we have

$$A = LU$$

$$\begin{bmatrix} 1 & -2 & 1 & 1 \\ 2 & -1 & 5 & -4 \\ -1 & 3 & -1 & 1 \\ -3 & 7 & -5 & 1 \end{bmatrix} = A = LU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 1/3 & 1 & 0 \\ -3 & 1/3 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2 & 1 & 1 \\ 0 & 3 & 3 & -6 \\ 0 & 0 & -1 & 4 \\ 0 & 0 & 0 & -6 \end{bmatrix}$$

In step 1, we performed:

- $\rho_2 \leftarrow \rho_2 - 2\rho_1$
- $\rho_3 \leftarrow \rho_3 - (-1)\rho_1$
- $\rho_4 \leftarrow \rho_4 - (-3)\rho_1$

Which is equivalent to $M_1 A$, where $M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{bmatrix}$

Note that $M_1^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 3 & 0 & 0 & 1 \end{bmatrix}$

In step 2, we performed

- $\rho_3 \leftarrow \rho_3 - (1/3)\rho_1$
- $\rho_4 \leftarrow \rho_4 - (1/3)\rho_1$

Which is equivalent to $M_2 M_1 A$, where $M_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1/3 & 1 & 0 \\ 0 & -1/3 & 0 & 1 \end{bmatrix}$

Note that $M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1/3 & 1 & 0 \\ 0 & 1/3 & 0 & 1 \end{bmatrix}$

In step 3, we performed

- $\rho_4 \leftarrow \rho_4 - 3\rho_1$

Which is equivalent to $M_3 M_2 M_1 A$, where $M_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -3 & 1 \end{bmatrix}$

Note that $M_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 3 & 1 \end{bmatrix}$

We can thus rearrange $U = M_3 M_2 M_1 A$ into $A = (M_3 M_2 M_1)^{-1} U = M_1^{-1} M_2^{-1} M_3^{-1} U = LU$

- The L and U for a matrix A are unique
- An alternate decomposition is $A = LD\hat{U}$, where $D = \text{diag}(U_{1,1}, \dots, U_{n,n})$ and $\hat{U}_{i,j} = \frac{U_{i,j}}{Ui,i}$
  - If A is symmetric, then $\hat{U} = L^{\mathrm{T}}$

Elementary Gauss Transformation: A matrix M corresponding to a single Gaussian row operation

- M is unit lower triangular
- M is non-zero at the diagonals (1s), and possibly in one column under the diagonal
- M is invertible and $M_{i,j}^{-1} = -M_{i,j}$
- $M_i = \mathbf{I} - M_i[:, i]\mathbf{e}_i^{\mathrm{T}}$ **and** $M_i^{-1} = \mathbf{I} + M_i[:, i]\mathbf{e}_i^{\mathrm{T}}$
- $M_1^{-1} \cdots M_k^{-1} = \mathbf{I} + \sum_{i=1}^{k} M_i[:, i]\mathbf{e}_i^{\mathrm{T}}$

GE/LU Algorithm		
Run GE on A without editing b. Get L and U.		• Asymptotically, same additions, multiplications, divisions as algorithm #1
$A\mathbf{x} = LU\mathbf{x} = L(U\mathbf{x}) = L\mathbf{v} = \mathbf{b}$	$\approx \frac{n^3}{3}$ flops, $\approx \frac{n^2}{2}$ divisions	
Using forward substitution, solve for **v in** $L\mathbf{v} = \mathbf{b}$	$\approx \frac{n^2}{2}$ flops, $\approx n$ divisions	• Useful for many linear systems with same A, different $\mathbf{b}$.
Using backward substitution, solve for **x in** $U\mathbf{x} = \mathbf{v}$	$\approx \frac{n^2}{2}$ flops, $\approx n$ divisions	• Since L has 1 as its pivots, technically no division needed

Choleski Factor: Of symmetric positive-definite matrix A, the lower triangular matrix $C = LD^{\frac{1}{2}}$, where D is diagonal

Choleski Factorization: For symmetric positive-definite matrix A, we have

$$A = CC^{\mathrm{T}} = LD^{\frac{1}{2}} \left(LD^{\frac{1}{2}}\right)^{\mathrm{T}}$$

Choleski Algorithm: A GE-based algorithm for computing C. Not in CSC336.

GE/LU Algorithm – (l, u)-Banded Case:
Since values with $i > j + l$ and $i < j - u$ are 0, we modify

```
for i in range(n - 1):
    for j in range(i + 1, min(i + 1 + l, n)):
```

each step to only modify a $(l+1)\times(u+1)$ submatrix.

Each step of LU preserves the matrix's bandedness; L is $(l,0)$-banded and U is $(0,u)$-banded

Forward/backward substitution on L and U, which are banded, require $n(l+u)$ flops together.

```
        A[j, i] /= A[i, i]
        for k in range(i + 1, min(i + 1 + u, n)):
            A[j, k] -= A[j, i] * A[i, k]
# This takes ≈ ∑_{i=1}^{n-1}(l+1)(u+1) ≈ ∑_{i=1}^{n} lu = nlu flops.
```

Inverse: Of square matrix $A \in \mathbb{R}^{n\times n}$, the matrix A^{-1} for which $A^{-1}A = AA^{-1} = \mathbf{I}$

Inverse Algorithm:

Let $\mathbf{x}_i$ be the i-th column of $A^{-1} = [\mathbf{x}_1 \quad \cdots \quad \mathbf{x}_n]$.

Break $AA^{-1} = \mathbf{I}$ column-wise, so $A\mathbf{x}_i = \mathbf{e}_i$:

$$\begin{bmatrix} A_{1,1} & \cdots & A_{1,i} & \cdots & A_{1,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{i,1} & \cdots & A_{i,i} & \cdots & A_{i,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{n,1} & \cdots & A_{n,i} & \cdots & A_{n.n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Solve the system for all $i \in \{1, \dots, n\}$. Find LU factorization of A first, which turns our systems into

$$L\mathbf{v}_i = \mathbf{e}_i$$
$$U\mathbf{x}_i = \mathbf{v}_i$$

Solve both equations using forward/back substitution.

```
L, U = LU_factor(A)
for i in range(n):
    v[i] = forward_substitution(L, e[i])
    x[i] = backward_substitution(U, v[i])
    A_inv[:, i] = x[i]
return A_inv

# LU factorization is n^3/3 flops, n^2/2 divisions
# F substitution is n^2/2 flops, n divisions
# S substitution is n^2/2 flops, n divisions
# Apply F/S substitution for all n systems
# Possible to reduce F sub from n^3/2 to n^3/6 flops
# Running-time is still O(n^3)
```

- We can find $A\mathbf{x} = \mathbf{b}$ by finding A^{-1} and calculating $A^{-1}\mathbf{b}$, but this is n^3 flops (3 times as much)

Pivoting and Scaling

Pivoting: The selection of a pivot x such that $x \neq 0$ and $|x| \gg 0$ to enhance usability and stability of GE.

- For LU factorization, where $P = P_n \cdots P_1$, where P_i is an elementary permutation matrix for step i of GE
  - $U = M_n P_n \cdots M_1 P_1 A$
  - Since $P_i^{-1} = P_i$, rearrange to $(M_n \cdots M_1)^{-1} U = LU = PA$
- **Row Pivoting ($P_r A$):** Reordering rows
- **Column Pivoting (AP_c):** Reordering columns
- **Partial Pivoting:** Reordering rows or columns
- **Complete Pivoting ($P_r A P_c$):** Reordering rows and columns
- **Symmetric Pivoting (PAP^{T}):** Reordering the same-indexed rows and columns

LU Algorithm – Row Pivoting:

For column i, instead of blindly choosing row i as the pivot (ie. cell $A_{i,i}$), choose row r_i such that

$$A_{r_i,i} = \max_{i' \in \{1,\dots,n\}} |A_{i',i}|$$
$$\Leftrightarrow r_i = \operatorname*{argmax}_{i' \in \{1,..,n\}} |A_{i',i}|$$

Choose big pivots – small pivots cause $\frac{A_{j,i}}{A_{i,i}} = \frac{A_{j,i}}{\text{pivot}}$ to be very big. When finding $L\mathbf{v} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{v}$, computing $b_i - \sum_{j=1}^{i-1} A_{i,j}x_j$ may lead to catastrophic collision.

To store permutations, use vector ipiv $\in \mathbb{N}^{n-1}$, where $\text{ipiv}_i = j \Leftrightarrow$ row i is swapped with row j in A

Note the cost is an extra $\sum_{i-1}^{n-1} i \approx \frac{n^2}{2}$ array comparisons.

```
# LU Factorization, Row Pivoting
for i in range(n - 1):
    # Find row index with highest column i value
    r_i = argmax(abs(A[i:, i]))
    # If column i is all 0, infinite solutions
    if A[r_i, i] == 0:
        return None
    # Swap rows
    A[i, :], A[r_i, :] = A[r_i, :], A[i, :]
    ipiv[i] = r_i

    # Rest of the algorithm
    for j in range(i + 1, n):
        A[j, i] /= A[i, i]
        A[j, i + 1:] -= A[j, i] * A[i, i + 1:]
```

eg. Solve $A\mathbf{x} = \mathbf{b}$ using three-decimal, base-10 floating-point arithmetic with rounding, where

$$A = \begin{bmatrix} -0.001 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, L = \begin{bmatrix} 1 & 0 \\ -1000 & 1 \end{bmatrix}, U = \begin{bmatrix} -0.01 & 1 \\ 0 & 1001 \end{bmatrix}$$

Solving $L\mathbf{v} = \mathbf{b}$,

$$\begin{bmatrix} 1 & 0 \\ -1000 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$v_1 = \frac{b_1}{L_{1,1}} = 1$$

$$v_2 = \frac{b_2 - L_{2,1}v_1}{L_{2,2}} = \frac{2 - \left(-1000(1)\right)}{1} = 1002$$

Solving $U\mathbf{x} = \mathbf{v}$,

$$\begin{bmatrix} -0.01 & 1 \\ 0 & 1001 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1002 \end{bmatrix}$$

$$x_2 = \frac{v_2}{U_{2,2}} = \frac{1002}{1001} \approx 1.001$$

$$x_1 = \frac{v_1 - U_{1,2}x_2}{U_{1,1}} = \frac{1 - 1\left(\frac{1002}{1001}\right)}{-0.01} = \frac{1000}{1001} \approx 0.999$$

Solving $L\mathbf{v} = \mathbf{b}$ computationally,

$$v_1 = 1$$
$$v_2 = 1002 \to 1000$$

Solving $U\mathbf{x} = \mathbf{v}$ computationally,

$$\begin{bmatrix} -0.01 & 1 \\ 0 & 1000 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1000 \end{bmatrix}$$

$$x_2 = \frac{v_2}{U_{2,2}} = 1, \qquad x_1 = \frac{v_1 - U_{1,2}x_2}{U_{1,1}} = \frac{1 - 1\left(\frac{1000}{1000}\right)}{-0.01} = 0$$

So we have $\mathbf{v} = \begin{bmatrix} 1 \\ 1002 \end{bmatrix}, \hat{\mathbf{v}} = \begin{bmatrix} 1 \\ 1000 \end{bmatrix}, \mathbf{x} \approx \begin{bmatrix} 1.001 \\ 0.999 \end{bmatrix}, \hat{\mathbf{x}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. It is clear there is catastrophic error in $\mathbf{x}, \hat{\mathbf{x}}$.

Instead, flip columns of A and b, compute with this:

$$A = \begin{bmatrix} 1 & 1 \\ -0.001 & 1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, L = \begin{bmatrix} 1 & 0 \\ -0.001 & 1 \end{bmatrix}, U = \begin{bmatrix} 1 & 1 \\ 0 & 1.001 \end{bmatrix}$$

Solving $L\mathbf{v} = \mathbf{b}$,

$$\begin{bmatrix} 1 & 0 \\ -0.001 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$v_1 = \frac{b_1}{L_{1,1}} = 2$$

$$v_2 = \frac{b_2 - L_{2,1}v_1}{L_{2,2}} = \frac{1 - (-0.001)(2)}{1} = 1.002$$

Solving $U\mathbf{x} = \mathbf{v}$,

$$\begin{bmatrix} 1 & 1 \\ 0 & 1.001 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1.002 \end{bmatrix}$$

$$x_2 = \frac{v_2}{U_{2,2}} = \frac{1.002}{1.001} \approx 1.001$$

$$x_1 = \frac{v_1 - U_{1,2}x_2}{U_{1,1}} = \frac{1 - 1\left(\frac{1.002}{1.001}\right)}{-0.01} \approx 0.999$$

Solving $L\mathbf{v} = \mathbf{b}$ computationally,

$$v_1 = 2$$
$$v_2 = 1.002 \to 1$$

Solving $U\mathbf{x} = \mathbf{v}$ computationally,

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$x_2 = \frac{v_2}{U_{2,2}} = 1, \qquad x_1 = \frac{v_1 - U_{1,2}x_2}{U_{1,1}} = \frac{2 - 1(1)}{1} = 1$$

So we have $\mathbf{v} = \begin{bmatrix} 2 \\ 1.002 \end{bmatrix}, \hat{\mathbf{v}} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{x} \approx \begin{bmatrix} 1.001 \\ 0.999 \end{bmatrix}, \hat{\mathbf{x}} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, which is the best we can get with 3-digit accuracy.

But, row pivoting still has its limits. Consider this:

$$A = \begin{bmatrix} -1 & 1000 \\ 1 & 1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1000 \\ 2 \end{bmatrix}, L = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}, U = \begin{bmatrix} -1 & 1000 \\ 0 & 1001 \end{bmatrix}$$

Solving $L\mathbf{v} = \mathbf{b}$, we get $v_1 = \frac{b_1}{L_{1,1}} = 1000, \quad v_2 = \frac{b_2 - L_{2,1}v_1}{L_{2,2}} = \frac{2-(-1)1000}{1} = 1002$

Solving $U\mathbf{x} = \mathbf{v}$, we get $x_1 = \frac{v_1 - U_{1,2}x_2}{U_{1,1}} = \frac{1000 - 1000\left(\frac{1002}{1001}\right)}{-1} = -\frac{1000}{1001} \approx -0.999, \quad x_2 = \frac{v_2}{U_{2,2}} = \frac{1002}{1001} \approx 1.001$

Solving $L\mathbf{v} = \mathbf{b}$ computationally, we get $v_1 = 1000, \quad v_2 = 1000$

Solving $U\mathbf{x} = \mathbf{v}$ computationally, we get $x_1 = \frac{v_1 - U_{1,2}x_2}{U_{1,1}} = \frac{1000 - 1000(1)}{-1} = 0, \quad x_2 = \frac{v_2}{U_{2,2}} = \frac{1000}{1000} = 1$

We will still get catastrophic error, even after flipping rows. To solve this, do **scaling** or **complete pivoting**!

LU Algorithm – Pivoting, Symmetric Case:

Row pivoting destroys symmetry – thus, we need symmetric pivoting.

```
# Swap rows and columns
A[i, :], A[r_i, :] = A[r_i, :], A[i, :]
A[:, i], A[:, r_i] = A[:, r_i], A[:, i]
```

```
ipiv[i] = r_i
```

LU Algorithm – Pivoting, Banded Case:

- Complete pivoting may destroy bandedness
- Row pivoting may make U $(0, u+l)$-banded; adds $\leq l$ non-zero superdiagonals
- Column pivoting may make L $(u+l, 0)$-banded; adds $\leq u$ non-zero subdiagonals

GE Algorithm – Row Pivoting:

Modify LU with pivoting to edit A and $\mathbf{b}$.
Using back substitution, solve for $\mathbf{x}$ in $A\mathbf{x} = \mathbf{b}$.

Same cost as the non-pivoting equivalents, plus the $\approx \frac{n^2}{2}$ comparisons from pivoting.

```
# Swap rows
A[i, :], A[r_i, :] = A[r_i, :], A[i, :]
b[i], b[r_i] = b[r_i], b[i]
# Rest of the algorithm
for j in range(i + 1, n):
    A[j, i] /= A[i, i]
    A[j, i + 1:] -= A[j, i] * A[i, i + 1:]
    b[j] -= A[j, i] * b[i]
```

GE/LU Algorithm – Row Pivoting:

Apply LU with pivoting to get L, U, ipiv from A

$$PA\mathbf{x} = LU\mathbf{x} = L(U\mathbf{x}) = L\mathbf{v} = P\mathbf{b}$$

Using forward substitution, solve for $\mathbf{v}$ in $L\mathbf{v} = P\mathbf{b}$
Using backward substitution, solve for $\mathbf{x}$ in $U\mathbf{x} = \mathbf{v}$

```
# Swap the rows of b; calculate Pb
for i in range(n - 1):
    b[i], b[ipiv[i]] = b[ipiv[i]], b[i]
v = forward_substitution(L, b)
x = backward_substitution(U, b)
```

Scaling: Practice of scaling rows so the largest element is 1. Improves stability of GE even more.

GE Algorithm – Scaling, Pivoting:

For all rows, find the maximum absolute value
Then divide the row by this value
Finally, apply GE with row pivoting.

This adds an extra n^2 comparisons and divisions.

```
for i in range(n):
    scale_factor = max(abs(A[i, :]))
    # If a row is all 0, then infinite solutions
    if scale_factor == 0:
        return None
    A[i, :] /= scale_factor
gaussian_elimination_row_pivot(A)
```

- LU factorization is $PDA = LU$, where $D_{i,i} = \frac{1}{\text{scale factor}} = \frac{1}{\max|A[i,:]|}$

GE Algorithm – Complete Pivoting:

Instead of for column i, finding the largest row, find the largest value below or right of the i-th diagonal.

Swap rows/columns until the max is the diagonal.

Dividing by the largest possible number means no need for scaling. Thus, complete pivoting improves accuracy on certain cases even further.

But...finding the argmax requires $\sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}$ comparisons, which is $\mathcal{O}(n^3)$ and less worth it than row pivoting/scaling, which is $\mathcal{O}(n^2)$ comparisons.

```
for i in range(n - 1):
    I, J = argmax(abs(A[i:, i:]))
    # If a square is all 0, infinite solutions
    if A[I, J] == 0:
        return None
    A[i, :], A[I, :] = A[I, :], A[i, :]
    A[:, i], A[:, J] = A[:, J], A[:, i]
    ipiv[i] = I
    jpiv[i] = J
    # Rest of the algorithm
    for j in range(i + 1, n):
        A[j, i] /= A[i, i]
        A[j, i + 1:] -= A[j, i] * A[i, i + 1:]
```

- LU factorization is $PAQ = LU$, where Q is the permutation matrix for swapped columns

Inner Product: A mapping $(S, S) \to \mathbb{R}$ for vector spaces S *(eg. dot product is inner product of vectors)*

- $(\mathbf{x}, \mathbf{x}) \geq 0$
- $(\mathbf{x}, \mathbf{x}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{0} \in S$
- $(\alpha\mathbf{x}, \mathbf{y}) = \alpha(\mathbf{x}, \mathbf{y})$
- $(\mathbf{x} + \mathbf{y}, \mathbf{z}) = (\mathbf{x}, \mathbf{z}) + (\mathbf{y}, \mathbf{z})$
- $(\mathbf{x}, \mathbf{y}) = (\mathbf{y}, \mathbf{x})$

Norm: A mapping $S \to \mathbb{R}^{\geq 0}$ that quantifies **distance/error** between vectors (i.e. $\|\mathbf{x} - \mathbf{y}\|$, or relative $\frac{\|\mathbf{x}-\mathbf{y}\|}{\|\mathbf{x}\|}$)

- $\|\mathbf{x}\| \geq 0$
- $\|\mathbf{x}\| = 0 \Leftrightarrow \mathbf{x} = \mathbf{0} \in S$
- $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$
- $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

p-norm/Holder Norm: $\|\mathbf{x}\|_p = \sqrt[p]{\sum_{i=1}^{n} |x_i|^p}$ $\quad \|A\|_p = \max_{\mathbf{x}\neq\mathbf{0}} \left\{\frac{\|A\mathbf{x}\|_p}{\|x\|_p}\right\} = \max_{\|\mathbf{x}\|_p=1} \{\|A\mathbf{x}\|_p\}$

Max/Infinity/Uniform Norm: $\|\mathbf{x}\|_\infty = \max_{i\in\{1,\dots,n\}} \{|x_i|\}$ $\quad \|A\|_\infty = \max_{i\in\{1,\dots,m\}} \left\{\sum_{j=1}^{n} |A_{i,j}|\right\}$ *("row norm")*

Euclidean Norm: $\|\mathbf{x}\|_2 = \sqrt[2]{\sum_{i=1}^{n} |x_i|^2}$

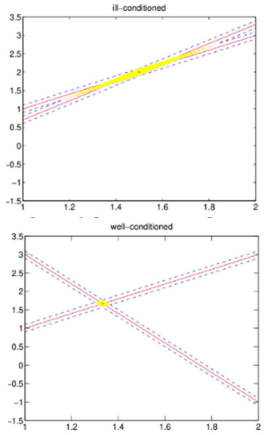
Manhattan/One Norm: $\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$ $\quad \|A\|_1 = \max_{j\in\{1,\dots,n\}} \{\sum_{i=1}^{m} |A_{i,j}|\}$ *("column norm")*

- $a \leq b \Rightarrow \|\mathbf{x}\|_a \leq \|\mathbf{x}\|_b$
- $\|\mathbf{I}\| = 1$
- $\|A\mathbf{x}\| \leq \|A\|\|\mathbf{x}\|$
- $\|AB\| \leq \|A\|\|B\|$
- $\|A\|_2^2 \leq \|A\|_1\|A\|_\infty$
- $\frac{1}{\sqrt{n}}\|A\|_2 \leq \|A\|_1 \leq \sqrt{m}\|A\|_2$
- $\frac{1}{\sqrt{n}}\|A\|_\infty \leq \|A\|_2 \leq \sqrt{m}\|A\|_\infty$
- $\frac{1}{m}\|A\|_1 \leq \|A\|_\infty \leq n\|A\|_1$

Cauchy-Schwarz Inequality: $|(\mathbf{x}, \mathbf{y})| \leq \sqrt{(\mathbf{x}, \mathbf{x})}\sqrt{(\mathbf{y}, \mathbf{y})}$

- $(\mathbf{x}, \mathbf{y}) = \|\mathbf{x}\|_2\|\mathbf{y}\|_2 \cos\theta$
- $|(\mathbf{x}, \mathbf{y})| = \|\mathbf{x}\|_2\|\mathbf{y}\|_2$

Holder Inequality: $\frac{1}{p} + \frac{1}{q} = 1 \Rightarrow |(\mathbf{x}, \mathbf{y})| \leq \|\mathbf{x}\|_p\|\mathbf{y}\|_q$



Residual: Of equation $A\mathbf{x} = \mathbf{b}$, value $r = \hat{\mathbf{b}} - A\hat{\mathbf{x}}$, all cumulative errors

Condition Number: Of invertible matrix A, the value $\kappa_a(A) = \|A\|_a\|A^{-1}\|_a$

- "Ratio of largest relative stretching to largest relative shrinking that the transformation can do to any non-zero vectors"
- "Relative sensitivity of solution x to $A\mathbf{x} = \mathbf{b}$ to changes in A and $\mathbf{b}$"
- "How close A is to being non-invertible"

- If A is not invertible, define $\kappa_a(A) = \infty$
- $\kappa_a(A) \geq 1$
- $\kappa_a(\mathrm{I}) = 1$
- A is orthogonal $\Rightarrow \|Q\|_2 = 1$
- A is square, orthogonal $\Rightarrow \kappa_2(A) = 1$
- $\frac{\|x-\hat{x}\|_a}{\|x\|_a} \leq \kappa_a(A)\left(\frac{\|b-\hat{b}\|_a}{\|b\|_a} + \frac{\|A-\hat{A}\|_a}{\|A\|_a} + \frac{\|r\|_a}{\|b\|_a}\right)$
- $\kappa_a^{-1}(A)\frac{\|r\|}{\|b\|} \leq \frac{\|x-\hat{x}\|_a}{\|x\|_a} \leq \kappa_a(A)\frac{\|r\|}{\|b\|}$

Well-Conditioned: Matrix A, if $\kappa_a(A) \approx 1$

Ill-Conditioned: Matrix A, if $\kappa_a(A) \approx \frac{1}{\epsilon_{\text{mach}}}$

- $\kappa_a(A) \geq 10^d$ in computer system with d decimal digits, solution may have 0 correct digits
- $\kappa_a(A) = 10^d$ in computer system with $e - d$ decimal digits, solution expected to have $\geq e - d$ correct digits
- Small r, large $x - \hat{x}$ is possible (eg. when $\kappa_a(A)$ is large)
- $\frac{1}{n}\kappa_2(A) \leq \kappa_1(A) \leq n\kappa_2(A)$
- $\frac{1}{n}\kappa_\infty(A) \leq \kappa_2(A) \leq n\kappa_\infty(A)$
- $\frac{1}{n^2}\kappa_1(A) \leq \kappa_\infty(A) \leq n^2\kappa_1(A)$

Nonlinear Systems

Nonlinear System: A system of form $f(\mathbf{x}) = 0$ for a nonlinear $f: \mathbb{R}^n \to \mathbb{R}^m$; our focus is $\mathbb{R}^n \to \mathbb{R}^n$.

Multiplicity: Of solution/root $\mathbf{x}$ to f, value k where $f^{(i)}(\mathbf{x}) = 0$ for $i < k$ and $f^{(k)}(\mathbf{x}) \neq 0$

- **Simple Root:** Solution $\mathbf{x}$ if its multiplicity $k > 1$
- **Multiple Root:** Solution $\mathbf{x}$ if its multiplicity $k = 1$

Fixed Point: Of function $f: \mathbb{R}^n \to \mathbb{R}^n$, point $\mathbf{x}$ if $f(\mathbf{x}) = \mathbf{x}$

Contractive/Systolic: In set $S \subseteq \mathbb{R}^n$, function $f: \mathbb{R}^n \to \mathbb{R}^n$ if $\exists \lambda \in [0,1), \forall \mathbf{x}, \mathbf{y} \in S, \|f(\mathbf{x}) - f(\mathbf{y})\| \leq \lambda \|\mathbf{x} - \mathbf{y}\|$

- $f: \mathbb{R} \to \mathbb{R}$ differentiable on $S \Rightarrow |f'(x)| \leq \lambda$
- Can be more convenient to construct $g: \mathbb{R}^n \to \mathbb{R}^n$ such that $f(\mathbf{x}) = 0 \Leftrightarrow g(\mathbf{x}) = \mathbf{x}$
  - $f: \mathbb{R} \to \mathbb{R}$ continuous on $[a,b]$, $f(a)f(b) < 0$ $\Rightarrow \exists x \in (a,b), f(x) = 0$
  - $g: \mathbb{R} \to \mathbb{R}$ continuous on $[a,b]$, $g([a,b]) \in [a,b]$ $\Rightarrow \exists x \in [a,b], g(x) = x$
  - $f: \mathbb{R} \to \mathbb{R}$ differentiable on (a,b), $f'(x) \neq 0, \exists x \in (a,b), f(x) = 0$ $\Rightarrow \exists x \in (a,b), f(x) = 0$ unique
  - $g: \mathbb{R} \to \mathbb{R}$ is contractive in $[a,b]$, $g([a,b]) \in [a,b]$ $\Rightarrow \exists x \in [a,b], g(x) = x$ unique

eg. $f(x) = x + \ln x$

Some functions g with $f(x) = 0 \Leftrightarrow g(x) = x$ are

$g_1(x) = -\ln x$

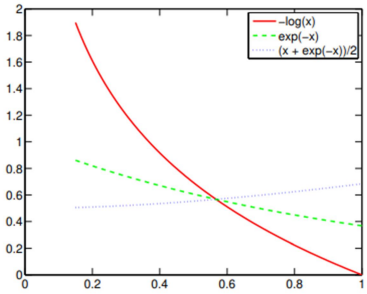
$$\begin{aligned} x &= g_1(x) \\ &= -\ln x \\ x + \ln x &= 0 \\ f(x) &= 0 \end{aligned}$$

$g_2(x) = e^{-x}$

$$\begin{aligned} x &= g_2(x) \\ &= e^{-x} \\ \ln x &= -x \\ x + \ln x &= 0 \\ f(x) &= 0 \end{aligned}$$

$g_3(x) = (x + e^{-x})/2$

$$\begin{aligned} x &= g_3(x) \\ &= (x + e^{-x})/2 \\ x &= e^{-x} \\ x + \ln x &= 0 \\ f(x) &= 0 \end{aligned}$$



Nonlinear Solver: An iterative method for solving a nonlinear equation $f(\mathbf{x}) = 0$. Usually starts with initial guess $\mathbf{x}_0$ and repeatedly iterates to values $\mathbf{x}_k$ that approach $\mathbf{x}$.

- **Residual:** Of nonlinear solver f at $\mathbf{x}_k$, the value $f(\mathbf{x}_k)$; its absolute/relative norms are $\|f(\mathbf{x}_k)\|$ and $\frac{\|f(\mathbf{x}_k)\|}{\|f(\mathbf{x}_0)\|}$
- **Stopping Criterion:** A criterion to stop iterating. Common ones include:
  - $\frac{\|f(\mathbf{x}_k)\|}{\|f(\mathbf{x}_0)\|} \leq \epsilon$
  - $\frac{\|\mathbf{x}_{k+1} - \mathbf{x}_k\|}{\|\mathbf{x}_{k+1}\|} \leq \epsilon$
  - $\|f(\mathbf{x}_k)\| \leq \epsilon$
  - $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| \leq \epsilon$
- $|f(x_k)| \leq \epsilon \Rightarrow |x_k - x| \approx \frac{\epsilon}{|f'(x)|}$. Thus, if $|f'(x)| \approx 0$, final residual can still be inaccurate.
- Smaller ϵ needed for slow converging methods

Nonlinear Solver, General Form

```
# Assume x is an array storing iterated values
guess x_0  # and maybe x_{-1}, …
for i in range(1, max_iterations):
    compute x_i  # maybe using x_{i-1} or f(x_{i-1})
    if stopping_criterion:
        break
```

One-Step: A method, if finding $\mathbf{x}_k$ only needs $\mathbf{x}_{k-1}$

Two-Step: A method, if finding $\mathbf{x}_k$ needs $\mathbf{x}_{k-1}, \mathbf{x}_{k-2}$

Converges: To x with order β, sequence $\{a_n\}_{n\in\mathbb{N}}$ if $\lim_{k\to\infty}\frac{\|a_{k+1}-x\|}{\|a_k-x\|^\beta}=\kappa$ for some **asymptotic error constant** $\kappa>0$

β	κ	Convergence	Example	
1	1	Sublinear	$a_n=\frac{1}{n}$,	then $\frac{\lvert a_{n+1}\rvert}{\lvert a_n\rvert}=\frac{n}{n+1}\to 1$
1	<1	Linear	$a_n=2^{-n}$,	then $\frac{\lvert a_{n+1}\rvert}{\lvert a_n\rvert}=\frac{1}{2}$
1	0	Superlinear	$a_n=n^{-n}$,	then $\frac{\lvert a_{n+1}\rvert}{\lvert a_n\rvert}=\frac{n^n}{(n+1)^{n+1}}\to 0$
>1	>0	Superlinear at rate β		
2	>0	Quadratic	$a_n=2^{(-n^2)}$,	then $\frac{\lvert a_{n+1}\rvert}{\lvert a_n\rvert^2}=2^{n^2-2n-1}>0$

Bisection Method: Nonlinear solver for continuous $f\colon\mathbb{R}\to\mathbb{R}$ where $f(a)f(b)<0$, finding one zero in range $[a,b]$.

Algorithm, Bisection Method

For each iteration i, set $x_i\leftarrow\frac{a+b}{2}$

If $f(x_i)=0$, solution obtained, break!

If $\frac{|b-a|}{2}\le\epsilon$, stopping condition, break!

Otherwise, check $f(a)$ and $f(\frac{a+b}{2})$'s signs

If same signs, set $a\leftarrow\frac{a+b}{2}$

If opposite signs, set $b\leftarrow\frac{a+b}{2}$

This halves $[a,b]$'s length and preserves $f(a)f(b)<0$.

```
x_{-1}, x_0 = a, b
for i in range(1, max_iterations):
    x_i = a + (b - a) / 2
    if abs(b - a) / 2 <= ϵ or f(x_i) == 0:
        break
    if (f(a) > 0) == (f(b) > 0):
        a = x_i
    else:
        b = x_i
```

- Converges linearly
- One function evaluation per iteration
- i-th iteration guarantees precision of $\frac{b-a}{2^i}$ to a root; though no guarantee to which root (if 2+ exist)
- Reaches precision of ϵ after $\le\lceil\log_2\frac{b-a}{\epsilon}\rceil$ iterations

Fixed-Point Iteration: One-step nonlinear solver, given function $g\colon\mathbb{R}^n\to\mathbb{R}^n$ where $f(\mathbf{x})=0\Leftrightarrow g(\mathbf{x})=\mathbf{x}$

Algorithm, Fixed-Point Iteration

Essentially calculates $(g\circ\cdots\circ g)(x_0)$. Relies on the fact that...

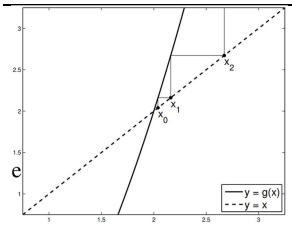
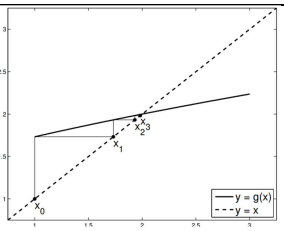
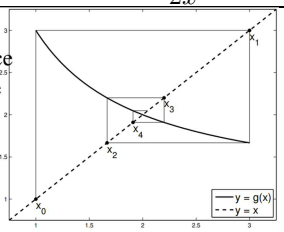
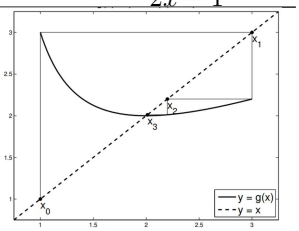
$g\colon\mathbb{R}\to\mathbb{R}$ is contraction on $[a,b]$ and $g([a,b])\in[a,b]\Rightarrow$

- $\exists x\in[a,b], g(x)=x$ unique,
- $\forall x_0\in[a,b], (g\circ\cdots\circ g)(x_0)\to x$,
- $|x_i-x|\le\lambda^i|x_0-x|\le\lambda^i\max\{x_0-a,b-x_0\}$

```
guess x_0
for i in range(1, max_iterations):
    x_i = g(x_{i-1})
    if stopping_criterion:
        break
```

If...	Then...
$\exists x\in\mathbb{R}$, ➢ $g(x)=x$ ➢ g' continuous on $(x-\epsilon,x+\epsilon)$ ➢ $\lvert g'(x)\rvert<1$	$\exists(c,d)\subseteq\mathbb{R}$, ➢ $x\in(c,d)$ ➢ $\forall x_0\in(c,d), (g\circ\cdots\circ g)(x_0)\to x$
$\exists x\in\mathbb{R}$, ➢ $g(x)=x$ ➢ $\forall x_0\in\mathbb{R}, (g\circ\cdots\circ g)(x_0)\to x$ ➢ g is C^β (expand!!!) ➢ $g^{(i)}(x)=0$ for $i<\beta$ ➢ $g^{(\beta)}(x)\neq 0$	➢ Rate of convergence of $(g\circ\cdots\circ g)(x_0)$ is β ➢ $\kappa=\frac{1}{\beta!}\lvert g^{(\beta)}(x)\rvert$

eg. $f(x) = x^2 - x - 2$

$g(x) = x^2 - 2$	$g(x) = \sqrt{x+2}$	$g(x) + \frac{1}{2x}$	$g(x) = \frac{x^2+2}{2x-1}$
			
Divergence (*g not contractive*)!	Monotone convergence	Alternating convergence	Quadratic convergence

Newton-Raphson Method: One-step non-zero solver for some root of differentiable $f: \mathbb{R}^n \to \mathbb{R}^n$ with $f'(x) \neq 0$.

Algorithm, Newton Method, $\mathbb{R} \to \mathbb{R}$

For each iteration i, find tangent line to f at x_i,

$$y = f(x_i) + f'(x_i)(x - x_i)$$

Solve the tangent line.

$$f(x_i) + f'(x_i)(x - x_i) = 0$$

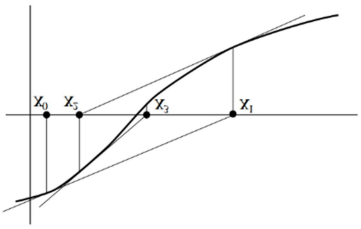
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Define $g(x) = x - \frac{f(x_i)}{f'(x_i)}$, then the algorithm becomes identical to fixed-point iteration.

```
guess x_0
for i in range(1, max_iterations):
    x_i = x_{i-1} - f(x_{i-1}) / f'(x_{i-1})
    if abs(f(x_i)) <= ϵ:
        break

# Finding derivative is computationally expensive
# Modified version – store and use f'(x_0)
# But convergence is much slower
# Compromise – evaluate the derivative occasionally
```

- Not guaranteed to converge, but does (and usually quadratically) if f twice differentiable, x_0 is "close enough" to x.
- Two function evaluations per iteration
- Converges slower to a multiple root, linearly with $\kappa = 1 - \frac{1}{m}$ for root multiplicity m. Some modifications can improve convergence.



Algorithm, Newton Method, $\mathbb{R}^n \to \mathbb{R}^n$

For each iteration i, find tangent line of f at $\mathbf{x}$,

$$y = f(\mathbf{x}_i) + Jf(\mathbf{x}_i)(\mathbf{x} - \mathbf{x}_i)$$

Solve the tangent line.

$$f(\mathbf{x}_i) + Jf(\mathbf{x}_i)(\mathbf{x} - \mathbf{x}_i) = 0$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i - [Jf(\mathbf{x}_i)]^{-1} f(\mathbf{x}_i)$$

This requires $Jf(\mathbf{x}_i)$ to be invertible.

```
guess x_0
for i in range(1, max_iterations):
    v_{i-1} = J(x_{i-1}) \ -f(x_{i-1})  # \ means solve system
    x_i = x_{i-1} + v_{i-1}
    if norm(f(x_i)) <= ϵ:
        break
```

- Same convergence properties as single-dimension Newton's method
- Computing the Jacobian is n^2 iterations, and each linear system is $\frac{n^3}{3}$ flops, a high cost
- Since the cost is high, there exist variants of this method ("quasi-Newton methods") to reduce cost – usually approximating the Jacobian without direct evaluation, avoiding solving the linear system, etc.

Secant Method: Two-step non-zero solver for some root of continuous $f: \mathbb{R} \to \mathbb{R}$

Algorithm, Newton Method

For each iteration i, find the line that passes $(x_i, f(x_i))$ and $(x_{i-1}, f(x_{i-1}))$,

$$y = f(x_i) + \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}(x - x_i)$$

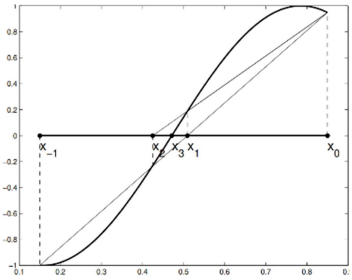
Solve the tangent line.

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

Note we need $f(x_i) \neq f(x_{i-1})$.

```
guess x_{-1}, x_0
for i in range(1, max_iterations):
    # Assume we store f(x_i) and x_i values
    x_i = x_{i-1} - f(x_i) * (x_i - x_{i-1}) / (f(x_i) - f(x_{i-1}))
    if abs(f(x_i)) <= ϵ:
        break
```

- Not always converges, but if so, at rate $\beta = \frac{1+\sqrt{5}}{2} \approx 1.618$, slower than Newton and faster than bisection.
- One function evaluation per iteration
- Converges slower to a multiple root
- Broyden's method is a quasi-Newton method considerable as a multivariate extension of secant method *(outside of CSC336's scope)*



Interpolation

Interpolant: Of function $f: \mathbb{R} \to \mathbb{R}$, function g where $g(x_i) = f(x_i)$ for finitely many $x_i \in \mathbb{R}$

- f interpolates $g \Leftrightarrow g$ interpolates f

Polynomial Approximation: Interpolation of complicated functions with polynomials

- Polynomials are easy to evaluate/differentiate/integrate, and are effective for continuous functions

Polynomial: Of degree n, a function of form $p_n(x) = \sum_{i=0}^{n} a_i f_i(x)$, where $f_i(x)$ is the basis of the polynomial.

Monomial Basis: The default basis $f_i(x) = x^i$, so $p_n(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$

Algorithm, Polynomial Evaluation, Naive

Just add up the terms, one-by-one.

Inefficient – n flops and exponentiations.

```
def polynomial_evaluate(a, x)
    return sum(a[i] * (x ** i) for i in range(n))
```

- For reference, $p_n'(x) = a_1 + 2a_2 x + 3a_3 x^2 + \cdots + n a_n x^{n-1}$

Weierstrass Theorem: f continuous on $[a, b] \Rightarrow \forall \epsilon > 0, \exists p_n(x)$ polynomial, $\max_{x \in [a,b]} |f(x) - p_n(x)| < \epsilon$

- Degree of p_n depends on ϵ – higher degrees increase cost, instability
- We are not given p_n, nor if p_n is an interpolant, so the theorem sucks. Only has theoretical value.

Horner's Rule: An efficient algorithm for polynomial evaluation

Algorithm, Polynomial Evaluation, Naive

Factor and calculate the polynomial as such:

$$\begin{aligned} p_n(x) &= a_0 + x(a_1 + a_2 x + \cdots + a_n x^{n-1}) \\ &= a_0 + x\big(a_1 + x(a_2 + \cdots + a_n x^{n-2})\big) \\ &= a_0 + x\left(a_1 + x(a_n + \cdots + x(a_{n-1} + x(a_n)))\right) \end{aligned}$$

More efficient – n flops.

```
def horner_rule_polynomial(a, x)
    y = a[n]
    for i in range(n - 1, -1, -1):
        y = y * x + a[i]
    return y
```

- Can be extended to derivative: $p_n'(x) = a_1 + x\left(2a_2 + x\left(3a_3 + \cdots + x\big((n-1)a_{n-1} + x(na_n)\big)\right)\right)$

Vandermonde Matrix: Matrix X in linear system $X\mathbf{a} = \mathbf{b}$, using $p_n(\mathbf{x})$ to interpolate n points

$$p_n(\mathbf{x}) = \mathbf{b} \Leftrightarrow \begin{cases} a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n = b_0 \\ \vdots \\ a_0 + a_1 x_n + a_n x_n^2 + \cdots + a_n x_n^n = b_n \end{cases} \Leftrightarrow \begin{bmatrix} x_0^0 & \cdots & x_0^n \\ \vdots & \ddots & \vdots \\ x_n^0 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} b_0 \\ \vdots \\ b_n \end{bmatrix}$$

- Adding new data requires recomputation of all a_i
- Ill-conditioned problem for large n – as such, technique not widely used in practice
- b_i are unique $\Leftrightarrow$ X is invertible

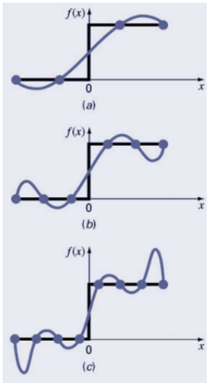
Spread ($\mathbf{spr}(S)$): Of a set S, the smallest interval $I \supseteq S$
Open Spread ($\mathbf{ospr}(S)$): Of a set S, the largest open interval $I \subseteq \text{spr}(S)$

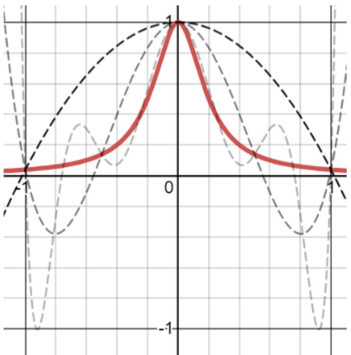
Basis	**Lagrange**	**Newton**
Notation	$l_j^{(n)}(x), l_j(x)$	$b_j^{(n)}(x), B_j(x)$
Closed Form	$\dfrac{\prod_{i=0;i\neq j}^{n}(x - x_i)}{\prod_{i=0;i\neq j}^{n}(x_j - x_i)}$	$\begin{cases} \prod_{i=0}^{j-1}(x - x_i) & j > 0 \\ 1 & j = 0 \end{cases}$
Example	Suppose x_i values are $\{1,3,4,5\}$, $l_1(x) = \dfrac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} = \dfrac{1}{4}(x-1)(x-4)(x-5)$	Suppose x_i values are $\{1,3,4,5\}$, $b_3(x) = (x-x_0)(x-x_1)(x-x_2) = (x-1)(x-3)(x-4)$
Polynomial	$p_n(x_i) = \sum_{j=0}^{n} a_j l_j(x_i)$	$p_n(x_i) = \sum_{j=0}^{n} a_j b_j(x_i)$
Properties	➢ Each term has order n ➢ $l_j(x_i) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$ ➢ $p_n(x)$ is very messy to compute, differentiate and integrate ➢ Adding data requires recomputing all a_i	➢ Each term has order j ➢ $i < j \Rightarrow b_j(x_i) = 0$ ➢ $b_j(x) = b_{j-1}(x)(x - x_j)$ ➢ $p_n(x)$ is easy to compute, messy to differentiate and integrate ➢ Adding data preserves existing a_i
Coefficient Computation	$f(x_i) = p_n(x_i) = \sum_{j=0}^{n} a_j l_j(x_i) = a_i$ Calculating a_i is trivial.	$f(x_i) = p_n(x_i) = \sum_{j=i}^{n} a_j b_j(x_i)$ Calculating a_i requires an algorithm (below).

- Lagrange/Newton bases are equivalent to monomial basis – the only difference is the computation
- Newton is usually the best choice for larger, complex systems
- Lagrange is undefined if $x_i = x_j$.
- Lagrange basis can prove with $n+1$ distinct data points, there exists a unique interpolant of degree $\leq n$
  - Suppose $p_n(x)$ and $q_n(x)$ interpolate the data, then $p_n(x_i) = q_n(x_i) = f(x_i)$
  - Define $r(x) = p_n(x) - q_n(x)$, so $\deg r(x) \leq n$
  - But $r(x_i) = 0$ for $i \in \{0, \ldots, n\}$, so r has $\geq n+1$ roots
  - The only way for $\deg r(x) \leq n$ and $r(x)$ has $\geq n+1$ roots is $r(x) = 0$, so $p_n(x) = q_n(x)$
- If f is C^{n+1} and $p_n(x)$ is a degree $\leq n$ polynomial interpolating f at $n+1$ distinct points,

$$\forall x \in \mathbb{R}, \exists \xi \in \text{ospr}\{x_0, \ldots, x_n, x\}, f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n}(x - x_j)$$

  - The error is 0 if $f^{(n+1)}(x) = 0$ or $x = x_i$





- If we can bound $|f^{n+1}(x)|$ and $\left|\prod_{j=0}^{n}(x-x_j)\right|$, we can bound error
- Many factors affect $W(x) = \left|\prod_{j=0}^{n}(x-x_j)\right|$
  - If the x_i are evenly spaced, $W(x)$ is higher near endpoints, smaller near midpoints.
  - **Runge's Phenomenon:** If n is large, $p_n(x)$ oscillates near edges of intervals if x_i are evenly-spaced. See $\frac{1}{1+25x^2} \rightarrow$
  - In general, large n is undesirable due to computational inefficiency and instability
- **Chevyshev Point:** A set of x_i denser near endpoints, sparser near mid-points in order to minimize $\max\left|\prod_{j=0}^{n}(x-x_j)\right|$ to minimize error bounds.

Newton Divided Differences: Method/algorithm for calculating interpolation coefficients a_i for Newton basis.

$$f(x_0) = p_n(x_0) = a_0$$
$$f(x_1) = p_n(x_1) = a_0 + a_1(x_1 - x_0)$$
$$f(x_2) = p_n(x_2) = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1)$$

$$a_0 = f(x_0)$$
$$a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$
$$a_2 = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0}$$

Define the following:

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

→ Then we have $a_i = f[x_0, \dots, x_i]$

Use a table like below. The top value of every cell is a value of a_i. Fill in columns from previous ones.

x_0	$f[x_0]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, \dots, x_3]$	$f[x_0, \dots, x_4]$	...
x_1	$f[x_1]$	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	$f[x_1, \dots, x_4]$	...	
x_2	$f[x_2]$	$f[x_2, x_3]$	$f[x_2, x_3, x_4]$	...	$f[x_{n-4}, x_n]$	
x_3	$f[x_3]$	$f[x_3, x_4]$	...	$f[x_{n-3}, \dots, x_n]$		
x_4	$f[x_4]$	...	$f[x_{n-2}, x_{n-1}, x_n]$			
...	...	$f[x_{n-1}, x_n]$				
x_n	$f[x_n]$					

Computationally, interpret this table as solving a triangular linear system $X\mathbf{a} = \mathbf{f}$, where

$$X = \begin{bmatrix} b_0(x_0) & b_1(x_0) & \cdots & b_n(x_0) \\ b_0(x_1) & b_1(x_1) & \cdots & b_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ b_0(x_n) & b_1(x_n) & \cdots & b_n(x_n) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & x_1 - x_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n - x_0 & \cdots & \prod_{i=0}^{n-1}(x_n - x_i) \end{bmatrix}, \mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, \mathbf{f} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}$$

Solving the system is $\mathcal{O}(n^2)$, while calculating the polynomial for a single point is $\mathcal{O}(n)$. We often evaluate the polynomial for over n points. In general, this is too expensive – we use piecewise interpolation methods instead.

Knot/Node/Breakpoint/Gridpoint: Of interval $I = [a, b]$, a set of $n + 1$ points $\{x_0, \dots, x_n\}$ partitioning I into n subintervals such that $a = x_0 < x_1 < \cdots < x_n = b$.

Piecewise Polynomial: Of degree N with respect to knots x_i, a polynomial with $\deg p_n(x) \leq N$ on all (x_{i-1}, x_i),

$$p(x) = \begin{cases} \vdots \\ a_0^{(1)} + a_1^{(1)} x + \cdots + a_N^{(1)} x^N & \text{if } x_0 \leq x < x_1 \\ \vdots \\ a_0^{(n)} + a_1^{(n)} x + \cdots + a_N^{(n)} x^N & \text{if } x_{n-1} \leq x \leq x_n \end{cases}$$

Spline: A continuous piecewise polynomial that is often C^{N-1}

- Normally, piecewise polynomials have $(N + 1)n$ chooseable coefficients
- Continuity reduces $n - 1$ chooseable coefficients
- C^k reduces $(k + 1)(n - 1)$ chooseable coefficients

eg. For $l_i(x) = a_i + b_i x$, piecewise linear polynomials have the form:

$$p(x) = \begin{cases} \vdots \\ l_i(x) & \text{if } x_{i-1} \leq x < x_i \\ \vdots \end{cases}$$

Since $l_i(x)$ has 2 chooseable coefficients, there are $2n$ chooseable coefficients in total.

Linear splines introduce the further constraints

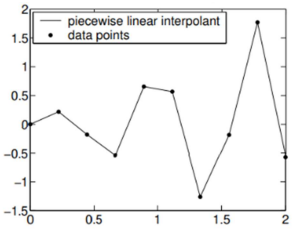
$$l_1(x_1) = l_2(x_1)$$
$$l_2(x_2) = l_3(x_2)$$
$$\vdots$$
$$l_{n-1}(x_{n-1}) = l_n(x_{n-1})$$

There are $n - 1$ equations which constrain $n - 1$ coefficients, so $2n - (n - 1) = n + 1$ chooseable coefficients.

Linear Spline: The continuous spline made by connecting neighbouring data points $(x_i, y_i), (x_{i-1}, y_{i-1})$ with lines.

$$L(x) = \begin{cases} \vdots \\ y_{i-1} \dfrac{x - x_i}{x_{i-1} - x_i} + y_i \dfrac{x - x_{i-1}}{x_i - x_{i-1}} & \text{if } x_{i-1} \leq x \leq x_i \\ \vdots \end{cases}$$

$$L'(x) = \begin{cases} \vdots \\ \dfrac{y_i - y_{i-1}}{x_i - x_{i-1}} & \text{if } x_{i-1} < x < x_i \\ \vdots \end{cases}$$



- L' is piecewise constant and continuous except possibly at knots
- If f is C^2, error satisfies $|f(x) - L(x)| \leq \frac{1}{8} \max_{x \in [a,b]} |f''(x)| \max_{i \in \{1,\dots,n\}} (x_i - x_{i-1})^2$
  - If $\deg f(x) \leq 1$, then $f''(x) = 0$ so the error is 0
- If $h = \max_{i \in \{1,\dots,n\}} \{x_i - x_{i-1}\}$, f'' is bounded in $[a, b]$, $\max_{x \in [a,b]} |f(x) - L(x)| = O(h^2)$
  - Error is **second order** – doubling n (ie. doubling data points, halving step sizes) divides error bound by 4
- Evaluating linear spline interpolant is $\mathcal{O}(1)$ – just find relevant subinterval, constant number of operations

Cubic Spline Interpolation: The unique smooth C^2 spline, where, for $C_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$

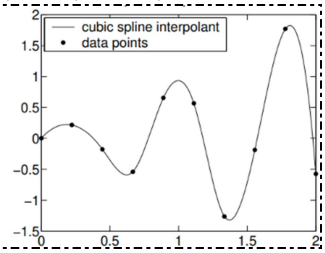
$$C(x) = \begin{cases} \vdots \\ C_i(x) & \text{if } x_{i-1} \leq x \leq x_i \\ \vdots \end{cases}$$

Since $C_i(x)$ has 4 chooseable coefficients, there are $4n$ chooseable coefficients in total.

To be smooth, we constrain

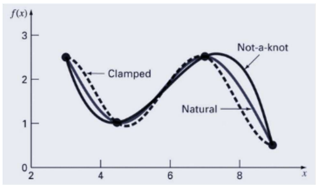
$$C_i(x_i) = C_{i+1}(x_i)$$
$$C_i'(x_i) = C_{i+1}'(x_i)$$
$$C_i''(x_i) = C_{i+1}''(x_i)$$

$3(n - 1)$ equations means $4n - (3n - 3) = n + 3$ chooseable coefficients.



- C''' is piecewise constant and continuous except possibly at knots
- **End/Boundary Condition:** Two extra constraints imposed at endpoints, because $C(x_i) = f(x_i)$ only uniquely determines $n + 1$ coefficients
  - *Natural/Free Cubic Spine:* Constraints $C''(x_i) = 0$ for $i \in \{0, n\}$
  - *Not-A-Knot Cubic Spine:* Constraints $C_i'''(x_i) = C_{i+1}'''(x_i)$ for $i \in \{1, n - 1\}$

- *Clamped Cubic Spine:* Constraints $C'(x_i) = f'(x_i)$ for $i \in \{0, n\}$
  - If $f'(x_0), f'(x_n)$ are given, just substitute the data
  - If $f'(x_0), f'(x_n)$ are not given, approximate them with a cubic polynomial interpolant on the first/last three data points



- If f is C^4, error satisfies
  $|f(x) - C(x)| \leq \frac{5}{384} \max_{x \in [a,b]} \left|f^{(4)}(x)\right| \max_{i \in \{1,\dots,n\}} (x_i - x_{i-1})^4$
  - If $\deg f(x) \leq 3$, then $f^{(4)}(x) = 0$ so the error is 0
- If $h = \max_{i \in \{1,\dots,n\}} \{x_i - x_{i-1}\}$, $f^{(4)}$ is bounded in $[a, b]$, $\max_{x \in [a,b]} |f(x) - C(x)| = O(h^4)$
  - Error is **fourth order** – doubling n (ie. doubling data points, halving step sizes) divides error bound by 16
- Constructing a cubic spline interpolant requires solving a $(n+1) \times (n+1)$ tridiagonal symmetric linear system ($\mathcal{O}(n)$). Evaluating the spline is $\mathcal{O}(1)$ – finding the right subinterval, constant-time operations

eg. Construction of clamped cubic spline interpolant

Consider $C_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$
Therefore $C_i'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2$
Therefore $C_i''(x) = 2c_i + 6d_i(x - x_i)$

Define $h_i = x_{i+1} - x_i$

Interpolating constraint is	$C_i(x_i) = f(x_i)$,	so $a_i = f(x_i)$
Continuity constraint is	$C_i(x_{i+1}) = C_{i+1}(x_{i+1})$,	so $a_{i+1} = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3$
C^1 constraint is	$C_i'(x_{i+1}) = C_{i+1}'(x_{i+1})$,	so $b_{i+1} = b_i + 2c_i h_i^2 + 3d_i h_i^3$
C^2 constraint is	$C_i''(x_{i+1}) = C_{i+1}''(x_{i+1})$,	so $c_{i+1} = c_i + 3d_i h_i$, AKA $d_i = \frac{c_{i+1} - c_i}{3h_i}$

Now, we need "base case" values so the rest of the equations follow from them.
For convenience, pretend there exists a $c_n = \frac{C''(x_n)}{2}$ so that the derivations of c_{i+1}, d_i hold for $i = n - 1$

We have, $C(x_n) = C_{n-1}(x_n) = f(x_n)$, so $a_{n-1} + b_{n-1}h_{n-1} + c_{n-1}h_{n-1}^2 + d_{n-1}h_{n-1}^3 = f(x_n)$
We have $C'(x_n) = C_{n-1}'(x_n) = f'(x_n)$, so $b_{n-1} + 2c_{n-1}h_{n-1} + 3d_{n-1}h_{n-1}^2 = f'(x_n)$
We have $C'(x_0) = C_0'(x_0) = f'(x_0)$, so $b_0 = f'(x_0)$

Do some fancy algebra, and end up with the tridiagonal, symmetric, strictly diagonally dominant system

$$\begin{bmatrix} 2h_0 & h_0 & 0 & \cdots & 0 \\ h_0 & 2(h_0 + h_1) & h_1 & \ddots & \vdots \\ 0 & h_1 & \ddots & h_{n-2} & 0 \\ \vdots & \ddots & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & \cdots & 0 & h_{n-1} & 2h_{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} \frac{3}{h_0}(f(x_1) - f(x_0)) - 3f'(x_0) \\ \frac{3}{h_1}(f(x_2) - f(x_1)) - \frac{3}{h_0}(f(x_1) - f(x_0)) \\ \vdots \\ \frac{3}{h_{n-1}}(f(x_n) - f(x_{n-1})) - \frac{3}{h_{n-2}}(f(x_{n-1}) - f(x_{n-2})) \\ -\frac{3}{h_{n-1}}(f(x_n) - f(x_{n-1})) + 3f'(x_n) \end{bmatrix}$$

Solving it returns c_i, which can be substituted to find d_i and b_i and a_i.

Shape-Preserving/Monotone Approximation: An approximation that keeps local extrema and derivative signs – for instance, $f(x_i) < f(x_{i+1}) \Rightarrow s(x)$ is increasing in (x_i, x_{i+1}), and $f(x_{i-1}) > f(x) < f(x_{i+1}) \Rightarrow s'(x_i) = 0$

- Includes linear spline interpolation and piecewise cubic hermite interpolation

Piecewise Cubic Hermite Interpolation: A C^1 shape-preserving version of cubic spline interpolation that also allows $f'(x)$ test data, where, for $s_i(x) = C_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$

$$s(x) = \begin{cases} \vdots \\ s_i(x) \quad \text{if } x_{i-1} \le x \le x_i \\ \vdots \end{cases}$$

Since $s_i(x)$ has 4 chooseable coefficients, there are $4n$ chooseable coefficients in total.

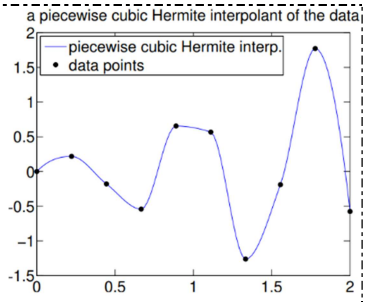
We constrain

$$s_i(x_i) = s_{i+1}(x_i)$$
$$s_i'(x_i) = s_{i+1}'(x_i)$$

$2(n-1)$ equations means $4n - (2n-2)$ $= 2n + 2$ chooseable coefficients.

$$s(x_i) = f(x_i)$$
$$s'(x_i) = f'(x_i)$$

These take up exactly all of them!



- Conditions can be shown to be linearly independent