Numerical Methods

Notes by Lev Leontev
Professor: Stephan Juricke

March 10, 2023

Contents

***Definition* 1** (Numerical Methods)**.** Numerical Methods are algorithmic approaches to numerically solve mathematical problems. We use them often when it is hard/difficult/impossible to solve a problem analytically.

1 Taylor series

Given a function $f : \mathbb{R} \to \mathbb{R}$ (that is hard to evaluate for some $x \in \mathbb{R}$), but f and $f^{(n)}$ are known for a value c, which is close to x. Can we use this information to approximate $f(x)$?

We know values for $\cos^{(n)}(0)$.

$$\begin{cases} f(0) = \cos(0) = 1 \\ f'(0) = -\sin(0) = 0 \\ f''(0) = -\cos(0) = -1 \end{cases} \qquad \text{for } c = 0$$

Can we get $\cos(0.1)$ from this?

***Definition* 1** (Taylor series)**.** Let $f : \mathbb{R} \to \mathbb{R}$, differentiable infinitely many times at $c \in \mathbb{R}$. So we have $f^{(k)}(c),\ k = 1, 2, \ldots$. Then the Taylor series of f at c is:

$$f(x) \approx f(c) + \frac{f(c)}{1!}(x-c)^1 + \frac{f''(c)}{2!}(x-c)^2 + \cdots = \sum_{k=0}^{\infty} \frac{f^{(k)}}{k!}(x-c)^k$$

Remark. Taylor series is a power series.

Remark. For $c = 0$ also known as Maclaurin series

Remark. A power series has an interval/radius of convergence. You can only evaluate the series if $x \in$ interval of convergence.

Example 1. What is the Taylor series for $f(x) = e^x$ at $c = 0$? We have $f^{(k)}(x) = e^x$, so $f^{(k)}(0) = 1$. Thus:

$$\sum_{k=0}^{\infty} \frac{1}{k!} x^k$$

and the radius of convergence is ∞.

I.e. for any $x \in \mathbb{R}$:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

For an algorithm we need a finite amount of terms. For example,

$$e^x \approx \frac{1}{0!}x^0 + \frac{1}{1!}x^1 + \frac{1}{2!}x^2 = 1 + x + \frac{x^2}{2}$$

This is a polynomial!

Example 2. Let's calculate Taylor series of a polynomial.

$$f(x) = 4x^2 + 5x + 7,\ c = 2$$
$$f(2) = 33,\ f'(2) = 8x + 5\Big|_{x=2} = 21,\ f''(2) = 8$$

Taylor series:

$$33 + 21(x-2) + \frac{8}{2}(x-2)^2 = 4x^2 + 5x + 7 = f(x)$$

Taylor series of a polynomial is itself.

Theorem 1 (Taylor theorem)**.** Let $f \in C^{n+1}([a,b])$ (i.e. f is $(n+1)$-times continuously differentiable). Then for any $x \in [a,b]$ we have that

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

where ξ_x is a point that depends on x and which is between c and x.

The first sum is called *truncated Taylor series*, the remainder is called *the error*.

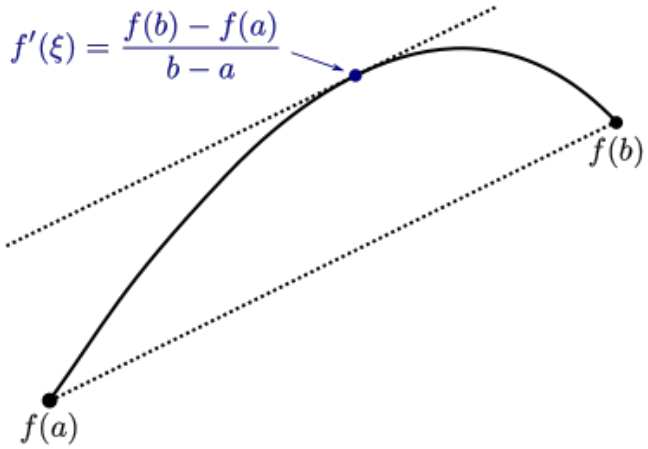
Example. For $n = 0$:

$$f(x) = f(c) + f'(\xi_x)(x-c)$$

Choose $c = a$, $x = b$:

$$f(b) = f(a) + f'(\xi_x)(b-a) \iff f'(\xi_x) = \frac{f(b)-f(a)}{b-a}$$

This is the mean value theorem!

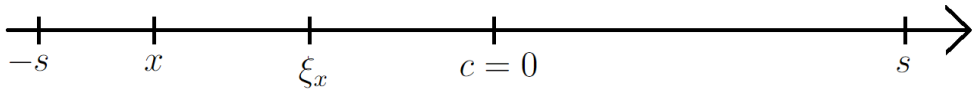


***Definition* 2.** We say that the Taylor series *represents* the function f at x if the Taylor series converges at that point, i.e. the remainder tends to zero as $n \to \infty$.

Example 1. Back to e^x: $f(x) = e^x$, $c = 0$, ξ_x is between c and x.

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!} + \frac{e^{\xi_x}}{(n+1)!}x^{n+1}$$

For any $x \in \mathbb{R}$ we find $s \in \mathbb{R}_0^+$ ($\mathbb{R}_0^+$ are all real, positive numbers including 0) so that $|x| \leqslant s$, and $|\xi_x| \leqslant s$ because ξ_x is between c and x.



Because e^x is monotone increasing, we have $e^{\xi_x} \leqslant e^s$, thus

$$\lim_{n\to\infty}\left|\frac{e^{\xi^x}}{(n+1)!}x^{n+1}\right| \leqslant \lim_{n\to\infty}\left|\frac{e^s}{(n+1)!}\right|s^{n+1} = e^s \lim_{n\to\infty}\frac{s^{n+1}}{(n+1)!} = 0$$

Because $(n+1)!$ will grow faster than any power of $s \implies \lim_{n\to\infty}\left|\frac{e^{\xi_x}}{(n+1)!}x^{n+1}\right| = 0$.

Thus e^x is *represented* by its Taylor series.

Example 2.

$$f(x) = \log(1+x),\ c = 0$$
$$f'(x) = \frac{1}{1+x} = (1+x)^{-1}$$
$$f''(x) = -(1+x)^{-2}$$
$$f'''(x) = +2(1+x)^{-3}$$
$$f^{(k)}(x) = (-1)^{k+1}(k-1)!\frac{1}{(1+x)^k}$$

So $f^{(k)}(0) = (-1)^{k-1}(k-1)!$ for $k \geqslant 1$, $f(0) = \log(1) = 0$.

Taylor series:

$$f(x) = \sum_{k=1}^{n} \frac{(-1)^{k-1}}{k}x^k + \frac{(-1)^k}{n+1}\frac{1}{(1+\xi_x)^{n+1}} \cdot x^{n+1} \quad \Big(\frac{n!}{(n+1)!} = \frac{1}{n+1}\Big)$$
$$E_n(x) = \frac{(-1)^k}{n+1}\frac{1}{(1+\xi_x)^{n+1}} \cdot x^{n+1} \text{ — the remainder}$$

Question: for which x does $\lim_{n\to\infty} E_n(x) = 0$?

$$\lim_{n\to\infty} E_n(x) = \lim_{n\to\infty} \frac{(-1)^n}{n+1}\Big(\frac{x}{\xi_x+1}\Big)^{n+1} \text{ for } \xi_x \in (c,x)\ (c=0)$$

Such a limit converges to 0, if the fraction is less than 1.

$$0 < \frac{x}{\xi_x+1} < 1 \iff x < \xi_x + 1 \iff x - \xi_x < 1 \text{ with } \xi_x \in (0,x) \iff x \leqslant 1$$

Consequence. $\lim_{n\to\infty} E_n(x) = 0$ if $0 < x \leqslant 1$. This means that the Taylor series represents $\log(x+1)$ for $x \in [0,1]$. We can extend this to show $x \in (-1,1]$.

Example 3. Let's compute $\cos(0.1)$. Let's approximate it with Taylor series with $c = 0$ (around zero).

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} \pm \cdots + \text{remainder}$$

Consequence.

$$\left|\cos(x) - \sum_{k=0}^{n}(-1)^k\frac{x^{2k}}{(2k!)}\right| = \left|(-1)^{n+1}\cos(\xi_x)\frac{x^{2(n+1)}}{\big(2(n+1)\big)!}\right| \leqslant \frac{0.1^{2(n+1)}}{2(n+1)!} \underset{n\to\infty}{\longrightarrow} 0$$

n	Taylor polynomial	$\lvert\text{error}\rvert \leqslant$
0	1	$\frac{(0.1)^2}{2} = 0.0005$
1	0.995	$\frac{0.0001}{24}$
2	0.99500416	$\frac{0.000001}{6!}$

Error depends on choice of $|x-c|$ and n.

Example 4. Compute $\log(2)$ using $f(x) = \log(x+1)$

$$\log(2) = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \ldots$$

Keeping 8 terms (until $n = 8$) we get $\log(2) \approx 0.63452$, the actual solution is $\log(2) = 0.693147$. Not so accurate. Can we improve?

We can use Taylor series of $\log\left(\frac{1+x}{1-x}\right)$ instead, since $\log\left(\frac{1+x}{1-x}\right) = \log(1+x) - \log(1-x)$. We choose $x = \frac{1}{3}$ instead of $x = 1$. Since x is closer to zero, both of the logarithms converge quicker.

$$\left(\log\left(\frac{1+1/3}{1-1/3}\right) = \log(2)\right)$$

We then get

$$\log(2) = 2 \cdot \left(\frac{1}{3} + \frac{1}{3^3 \cdot 5} + \dots\right)$$

We only need 4 terms to get $\log 2 \approx 0.69313$.

Theorem 2 (Reformulation of Taylor's theorem)**.** $f \in C^{n+1}([a,b])$. We change c to x and the old x to $x+h$ from previous version $\implies$ get for $x,\ x+h \in [a,b]$:

$$f(x+h) = \sum_{k=0}^{n} \frac{f^{(k)}(x)}{k!} h^k + \frac{f^{(n+1)}(\xi_x)}{(n+1)!} h^{n+1} \text{ where } \xi_x \in (x, x+h),\ h > 0$$

We can write error term as

$$f(x+h) - \sum_{k=0}^{n} \frac{f^{(k)}(x)}{k!} h^k = \mathcal{O}(h^{n+1})$$

Remark. Let's recall what the $\mathcal{O}$-notation means. $a(h) = \mathcal{O}\big(b(h)\big)$ if $\exists c > 0$ such that $\frac{a(j)}{b(j)} \leqslant c$ as $h \to 0$. So, for $n = 1$ the error decreases with h^2 (quadratic convergence).
$n = 2$: error decreases cubically, i.e. h^3, etc.

Summary of Taylor series:

- Problem: Evaluate $f(x)$ with a given error bound.
- Required: $f \in C^{n+1}$, values of derivatives $f^{(k)}(a)$.
- Check interval of convergence: does Taylor series expansion work?
- Estimate the maximum error for n terms of the Taylor polynomial.
- Choose n, such that the error bound is low enough.
- Evaluate the Taylor polynomial.

2 Number representation

2.1 Errors

There are different error types:

1. An error in data (partly due to roundoff).
2. Roundoff errors (during computation). For example, multiplication increases the amount of needed significant digits, and we can't store them all on a computer.
3. Truncation error, that is inherent to numeric methods. For example, if we take a finite number of terms in our Taylor series.

***Definition* 1.** Let $\tilde{a}$ be an approximation of a. Then $|\tilde{a} - a|$ is the *absolute error*, and $\left|\frac{\tilde{a}-a}{a}\right|$ is the *relative error*. The *error bound* is the magnitude of admissable error.

Example. 0.00123 with error $0.000004 = 0.4 \cdot 10^{-5}$. The error is below $\frac{1}{2} \cdot 10^{-t}$ with $t = 5$, so there's 5 correct digits and 3 significant digits (the number of non-leading zeros).

Example. 0.00123 with error $0.000006 = 0.06 \cdot 10^{-4} > \frac{1}{2} \cdot 10^{-5}$. Only has 2 significant digits, because we have to round the error up.

Theorem 1. In addition/subtraction the bounds for absolute errors are added, in multiplication/division the relative errors are added.

Example. Solve $x^2 - 56x + 1 = 0$:

$$x = 28 - \sqrt{783} \approx 28 - 27.982 \text{ (5 significant digits)} = 0.018 \pm \frac{1}{2} \cdot 10^{-3}$$

We end up with 2 significant digits in the answer, despite that we used to have 5. That's why computers use floating point numbers, as leading zeros are bad.

***Definition* 2** (Error propagation)**.** If $y(x)$ is smooth, than the derivative $|y(x)'|$ can be interpreted as the sensitivity of $y(x)$ to errors in x. We can generalize this to functions of multiple variables:

$$|\Delta y| \leqslant \sum \left|\frac{\partial y}{\partial x_i}\right| \cdot |\Delta x_i|, \text{ where } \Delta y = \tilde{y} - y,\ \Delta x_i = \tilde{x_i} - x_i$$

This is an empirical inequality that is only valid for small Δx_i. It is used a lot in physics.

2.2 Base representation

***Definition* 3** (Base representation)**.** Every number $x \in \mathbb{N}$ can be written in the following form as a unique expansion with respect to the base b, where $b \in \mathbb{N} \setminus \{1\}$:

$$x = a_0 b^0 + a_1 b^1 + a_2 b^2 + \cdots + a_n b_n = \sum_{i=0}^{n} a_i b^i$$
$$a \in \mathbb{N}_0,\ a_i < b,\ a_i \in \{0, \ldots, b-1\}$$

Here b is called the *base*, a_i are called the *digits*. Humans usually use base 10. But, for example, computers can use base 2.

For a real number $x \in \mathbb{R}$ we can write:

$$x = \sum_{i=0}^{n} a_i b^i + \sum_{i=1}^{\infty} a_{-i} b^{-i}$$

Example. $b = 2:\ 1011 = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = (11)_{10}$

There are different algorithms that convert number systems.

2.3 Euclid's algorithm

Euclid's algorithm converts $(x)_{10}$ to $(y)_b$.

1. Input $(x)_{10}$.
2. Determine the smallest n, such that $x < b^{n+1}$.
3. For $i = n$ to 0 do:

$$a_i \coloneqq x \text{ div } b^i \text{ (integer division)}$$
$$x \coloneqq x \bmod b^i \text{ (the remainder)}$$

4. Output result $a_n a_{n-1} a_{n-2} \dots a_0 = (y)_b$.

Example. 1. $(x)_{10} = (13)_{10} \to (y)_2$

2. $n = 3$ since $13 < 2^4$.
3.

$$i = 3:\ a_3 = 13 \text{ div } 2^3 = 1,\ x = 13 \bmod 2^3 = 5$$
$$i = 2:\ a_2 = 5 \text{ div } 2^2 = 1,\ x = 5 \bmod 2^2 = 1$$
$$i = 1:\ a_1 = 1 \text{ div } 2^1 = 0,\ x = 1 \bmod 2^1 = 1$$
$$i = 0:\ a_0 = 1 \text{ div } 2^0 = 1,\ x = 1 \bmod 2^0 = 0$$

4. Output: $(1101)_2 = (13)_{10}$.

Two problems of the Euclid's algorithm:

1. Step 2 is inefficient
2. Division by large numbers can be problematic.

2.4 Horner's scheme

Horner's scheme is a more efficient algorithm. The idea is to represent the number as follows:

$$(a_n a_{n-1} \dots a_0)_b = a_0 + b\big(a_1 + b(a_2 + b(a_3 + \dots + b(a_n))) \dots\big)$$

The algorithm is the following:

1. Input $(x)_{10}$.
2. $i \coloneqq 0$.

3. While $x > 0$ do:

$$a_i \coloneqq x \bmod b$$
$$x \coloneqq x \operatorname{div} b$$
$$i \coloneqq i + 1$$

4. Output result $a_n a_{n-1} a_{n-2} \dots a_0 = (y)_b$.

Remark. The algorithm is very similar to the Euclid's algorithm — the difference is that we execute it in reverse. We no longer have divisions by large numbers, and thus it runs faster.

General remarks:

- A number with simple representation in one base may be complicated to represent in another base. For example, $(0.1)_{10} = (0.0001100110011\dots)_2$.
- Base 2 is called *binary*, base 8 is *octal*, base 16 is *hexadecimal*.
- To convert from a base b to base 10 we can just perform the following computation:

$$(42)_8 = 4 \cdot 8^1 + 2 \cdot 8^0 = (34)_{10}$$

- Conversion 2 and 8. $8 = 2^3$: three consecutive bits represent one octal digit, e.g.

$$(551.624)_8 = (101101001.110010100)_2$$

- Conversion 2 and $16 = 2^4$: just four bits to one hexadecimal digit.
- Horner's scheme algorithm does not need estimate of n and does not divide by large numbers.
- It is applicable to real numbers, but one needs a critireon to stop if the representation with the new base is infinite.
- On computers we only have finite precision (the number of digits/bits).

2.5 Floating point representation

***Definition* 4.** Normalized floating point representation with respect to a base b stores any number x as follows:

$$x = 0.a_1 a_2 \dots a_k \cdot b^n \text{ where } a_i \in \{0, 1, \dots, b-1\}$$

a_i are called *digits*, k is called *precision*, n is called *exponent*, $a_1 \dots a_k$ is called *mantissa*, $a_1 \neq 0$ is called *normalization*, which makes the representation unique.

Remark. Leading zeros are essentially a waste of space. That's why floating points are useful when you add/subtract/divide numbers — as you're able to move the decimal point.

Example 1. Base 10: $32.213 = 0.32213 \cdot 10^2$.

Base 2: $x = \pm 0.b_1 b_2 \dots b_k \cdot 2^n$ We need another bit to define the sign of the number.

Example 2. For *single precision* floating-point numbers: 4 bytes = 32 bits.

- 1 bit sign mantissa.

- 1 bit sign exponent.
- 7 bits for exponent (integer).
- 23 bits for mantissa (24 effectively, since the first digit is always 1).

7 bits allow for the largest exponent of 127, i.e. $2^{127} \approx 10^{38}$. Anything above that is infinity. So, we have the range of $10^{-38} < |x| < 10^{38}$.

Since $2^{-24} \approx 10^{-7}$, we can represent 7 significant digits.

We can have a better representation, e.g. with *double precision* (8 bytes).

Issues with floating-point numbers:

- Adding numbers is not commutative, i.e. $x + y$ does not necessarily equal to $y + x$.
- It's not associative, i.e. $(x + y) + z$ does not necessarily equal to $x + (y + z)$. You usually try to add small numbers together first, in hopes that they will become big enough to become significant.

Example 1. Take $x = y = 0.00000033$, $z = 0.00000034$, $w = 1.00000000$. We compute $x + y + z + w$, and in that order we get 1.000001. Reversed order gets 1.000000 with base 10. Why is that? That's because we only have 7 significant digits.

Example 2.

$$\sum_{1}^{10^7} 1 + 10^7 = 1 + \cdots + 1 + 10^7$$

The order of summation will make a difference. You will either get 10^7 or $2 \cdot 10^7$, because $1 + 1 = 2$, but $1 + 10^7 = 10^7$ due to roundoff.

Consequence. Avoid adding numbers of different order of magnitude. Add numbers in increasing order of their size.

Example 3. Compute $x - \sin x$ for x close to 0, e.g. $x = 1/15$. Assume $k = 10$ precision.

$$\begin{aligned} x &= 0.6666666667 \cdot 10^{-1} \\ \sin x &= 0.6661729492 \cdot 10^{-1} \\ x - \sin x &= 0.0004937175 \cdot 10^{-1} = 0.4937175\underline{000} \cdot 10^{-4} \end{aligned}$$

Notice the three zeros at the end — that is a sign of a precision loss (unless the number actually ends with zeros).

Consequence. Avoid subtracting numbers of similar size, because it leads to a loss of precision.

A potential solution in this case would be to use Taylor series expanison:

$$\begin{aligned} \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \\ x - \sin x &= +\frac{x^3}{3!} - \frac{x^5}{5!} + \dots \end{aligned}$$

Using 3 terms we get: $0.4937174328 \cdot 10^{-4}$. The error of the Taylor series with 3 terms will be $\leqslant 10^{-13}$ (which is needed for "full" precision for $0.\ldots \cdot 10^{-4}$).

Theorem 2. Let x, y be two normalized floating point numbers with $x > y > 0$ and base $b = 2$. If there exist $p, q \in \mathbb{N}_0$ such that

$$2^{-p} \leqslant 1 - \frac{y}{x} \leqslant 2^{-q}$$

Then at most p and at least q significant bits (digits base 2) are lost during subtraction.

3 Linear Systems of Equations

***Definition* 1.** A linear system of equations is given by

$$Ax = b,\ A \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^n, b \in \mathbb{R}^m$$

I.e. the matrix A has m rows and n columns, x is a vector with n unknowns, b has m entries, thus the system has m equations.

Remark. The system of equations is called *linear*, because the degree of all x_i is equal to one.

Remark. If $n = m$, the system is called *square*. (As the matrix is square).

Remark. We can also write the system as a sum:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i,\ i = 1, \dots, m$$

Example.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases} \qquad n = 2,\ m = 2$$

Linear systems of equations arise in a lot of problems:

- Geometrical problems (coordinate transforms, 3D matrices).
- Electrical circuits, Kirchhoff's laws/Ohm's laws.
- Solving differential equations.
- GPS.

3.1 Gaussian Elimination (GE)

Assume that $m = n$ (square system). The idea of Gaussian Elimination: do row operations to produce an upper triangular matrix (echelon form). Then do backward substitution to solve the system.

Allowed row operations:

1. Swap rows.
2. Scale rows, i.e. multiply a row by a scaler.
3. Add multiples of one row to another.

Example.

$$A = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix}, \quad b = \begin{bmatrix} 16 \\ 26 \\ -19 \\ -34 \end{bmatrix}$$

Step 1. Do GE in a systematic way:

$$\text{Augmented matrix} = \left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 16 \\ 12 & -8 & 6 & 10 & 26 \\ 3 & -13 & 9 & 3 & -19 \\ -6 & 4 & 1 & -18 & -34 \end{array}\right]$$

The 6 here is the pivot element, and the first row is the pivot row.

$$\left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 16 \\ 12 & -8 & 6 & 10 & 26 \\ 3 & -13 & 9 & 3 & -19 \\ -6 & 4 & 1 & -18 & -34 \end{array}\right] \begin{array}{l} \leftarrow \text{ pivot row} \\ \leftarrow (-2)\cdot R_1 + R_2 \\ \leftarrow (-1/2)\cdot R_1 + R_3 \\ \leftarrow 1\cdot R_1 + R_4 \end{array}$$

$$\hookrightarrow \left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 16 \\ 0 & -4 & 2 & 2 & -6 \\ 0 & -12 & 8 & 1 & -27 \\ 0 & 2 & 3 & -14 & -18 \end{array}\right] \begin{array}{l} \\ \leftarrow \text{ pivot row} \\ \leftarrow (-3)\cdot R_2 + R_3 \\ \leftarrow (1/2)\cdot R_2 + R_4 \end{array}$$

Always consider the factor, e.g.

$$-3 = -\left(\frac{-12}{-4}\right)$$
$$\frac{1}{2} = -\left(\frac{2}{-4}\right)$$

Eventually, we end up with a triangular form (using diagonal elements as pivots).

$$\left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 16 \\ 0 & -4 & 2 & 2 & -6 \\ 0 & 0 & 2 & -5 & -9 \\ 0 & 0 & 0 & -3 & -3 \end{array}\right]$$

Step 2. Backward substitution:

- Last row: $-3x_4 = -3 \Longleftrightarrow x_4 = 1$.
- Second last row:

$$2x_3 - 5x_4 = -9$$
$$2x_3 - 5 = -9 \Longleftrightarrow x_3 = -2$$

- ... finally: $x_1 = 3,\ x_2 = 1,\ x_3 = -2,\ x_4 = 1$.

The algorithm again:

1. Input $A \in \mathbb{R}^{n\times n},\ b \in \mathbb{R}^n$.

   Forward substitution:

2. For $k = 1, \ldots, n-1$ (for all pivot rows, except the last one):

3. For $i = k+1, \ldots, n$ (for all rows below the pivot row):

4. For $j = k, \ldots, n$ (for all columns from the pivot one):
   $a_{ij} := a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj}$
   End for.
   $b_i := b_i - \frac{a_{ik}}{a_{kk}} b_k$

$\overline{3}$. End for.

$\overline{2}$. End for.

Backward substitution:

5. $x_n = \frac{b_n}{a_{nn}}$ (last unknown)

6. For $i = n-1, \ldots, 1$ (return back row by row)
   rhs $:= b_i$

7. For $j = n, \ldots, i+1$ (for all columns up to the pivot element)
   rhs $:=$ rhs $- a_{ij}x_i$ (x_j are already known)

$\overline{7}$. End for. $x_i := \frac{\text{rhs}}{a_{ii}}$

$\overline{6}$. End for.

GE can be used whenever the pivots don't vanish.

Example.

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_1 + x_2 + 2x_3 = 2 \\ x_1 + 2x_2 + 2x_3 = 1 \end{cases} \implies \begin{cases} x_1 = 1 \\ x_2 = -1 \\ x_3 = 1 \end{cases}$$

But addition of rows will give us:

$$\begin{cases} x_3 = 1 \text{ – here we have a missing pivot} \\ x_2 + x_3 = 0 \end{cases} \qquad \left(\begin{array}{ccc|c} 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 1 \end{array}\right)$$

We already get into trouble with very small pivot elements.

Example. Let $\varepsilon > 0$ and consider

$$\begin{cases} \varepsilon x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases} \iff \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \vec{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

For $\varepsilon \ll 1$, the actual solution is $x_1 \approx x_2 \approx 1$. However, GE yields

$$x_2 = \frac{2 - \frac{1}{\varepsilon}}{1 - \frac{1}{\varepsilon}} \overset{\varepsilon \ll 1}{\approx} \frac{-\frac{1}{\varepsilon}}{-\frac{1}{\varepsilon}} = 1$$

WIth finite precision we will get through backward substitution: $x_2 = 1$ and $x_1 = \frac{1-x_2}{\varepsilon} = 0$ which is wrong. The pivot is too small. But change order of equations.

$$\begin{cases} x_1 + x_2 = 2 \\ \varepsilon x_1 + x_2 = 1 \end{cases} \overset{\text{GE}}{\implies} \begin{cases} x_2 = \frac{1-2\varepsilon}{1-\varepsilon} \\ x_1 = 2 - x_2 \end{cases}$$

Now the answer is correct. The reason why the first one was incorrect is error amplification of x_2 by multiplication. $\frac{1}{\varepsilon}$ leads in the first case to a wrong result.

3.2 Scaled partial pivoting

***Definition* 2.** *Pivoting* means that the pivot element is chosen appropriately, and not just row by row.

***Definition* 3.** *Partial pivoting* means we will reorder rows (not columns, otherwise it would be full pivoting).

***Definition* 4.** *Scaled* means we look for best *relative* pivot, i.e. best ratio between pivot element and maximal entry of row (all in absolute values).

Remark. This will lead to minimal error propagation.

The algorithm:

1. Input $A \in \mathbb{R}^{n\times n},\ b \in \mathbb{R}^m$.
2. Find maximal absolute values of entries in rows $s \in \mathbb{R}^n$, such that $s_i = \max_{j=1}^{n} |a_{ij}|$.
   Forward elimination:
3. For $k = 1, \ldots, n-1$ (for all pivot rows).
4. For $i = k, \ldots, n$ (for all rows below pivot row)
   compute $\left|\frac{a_{ik}}{s_i}\right|$.

$\overline{4}$. End for.

5. Find row with the largest relative pivot element, name it row j.
6. Swap k with j.
7. Swap entries k and j in vector s.
8. Do skip of forward elimination in row k.

$\overline{3}$. End for.

Backward substitution is done as before, but with updated order.

Example.

$$\left[\begin{array}{cccc|c} 3 & -13 & 9 & 3 & -19 \\ -6 & 4 & 1 & -18 & -32 \\ 6 & -2 & 2 & 4 & 16 \\ -12 & -8 & 6 & 10 & 26 \end{array}\right]$$

Initial $s = (13, 18, 6, 12)$. Iterations:

1. • Relative pivots:
$$\left(\frac{3}{13}, \frac{6}{18}, \frac{6}{6}, \frac{12}{12}\right) = \left(\left|\frac{a_{ik}}{s_i}\right|\right)$$
   • Rows 3 and 4 have pivot 1 greater than all others. Select for swapping rows 1 and 3.

$$\left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 16 \\ -6 & 4 & 1 & -18 & -32 \\ 3 & -13 & 9 & 3 & -19 \\ 12 & -8 & 6 & 10 & 26 \end{array}\right]$$

- Swap entries $3 \leftrightarrow 1$ in s: $(6, 18, 13, 12)$.
- Forward elimination step (like in GE):

$$\left[\begin{array}{cccc|c} 6 & -2 & 2 & 4 & 16 \\ 0 & 2 & 3 & -14 & -18 \\ 0 & -12 & 8 & 1 & -27 \\ 0 & -4 & 2 & 2 & -6 \end{array}\right]$$

2. On the second iteration, $k = 2$.
   - Relative pivots (we don't care about the first row anymore, so just three rows left):

$$\left(\left|\frac{2}{18}\right|, \left|\frac{12}{13}\right|, \frac{4}{12}\right)$$

     The second ratio is the largest, and it corresponds to the third row.
   - So, we swap row 3 with row $k = 2$.
   - Swap entries in s.
   - Forward elimination. Then backward substitution on updated matrix as before.

Remarks:

- In efficient implementations, the step of row swapping can be omitted, just a permutation vector l needs to be stored to keep track of matrix rearrangements. This will result in "echelon form" that will look like e.g.

$$\begin{array}{cc} 2 & \to \\ 4 & \to \\ 1 & \to \\ 3 & \to \end{array} \left[\begin{array}{cccc|c} 0 & * & * & * & * \\ 0 & 0 & 0 & * & * \\ * & * & * & * & * \\ 0 & 0 & * & * & * \end{array}\right]$$

- GE with scaled partial pivoting always works when matrix is invertible, i.e. there exists a A^{-1}, such that $AA^{-1} = I$.

  It will fail for a singular (i.e. not invertible) matrix, because eventually a division by 0 will occur.

- Doing Gaussian elimination has computational complexity of $\mathcal{O}(n^3)$, because we have three nested for-loops. Cubic behaviour n^3 is problematic for large n!

- Traditionally, only the multiplication/division operations were counted in the number of operations C. (Since addition is very cheap). On present-day hardware, however, the costs are nearly as "cheap" as addition or subtraction.

- We are missing costs due to exchange with memory. Therefore, estimates of time complexity and reality may diverge substantially.

- Backward substitution has order n^2, which does not affect the general estimate of n^3.

- Scaled partial pivoting leads to an increase in cost, but order stays n^3.

3.3 Banded systems

***Definition* 5.** An equation system is called *banded* if $a_{ij} \neq 0, |i-j| \leqslant k < n$, i.e. non-zero entries are arranged around the diagonal.

Example.

$$\begin{pmatrix} * & * & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & * & * \end{pmatrix}$$

Is a 3-banded system with $k = 1$.

Remark. Banded systems can occur in image filtering and many other problems.

Remark. The case of $k = 1$ is called triagonal or three diagonal.

Remark. Solving Gaussian Elimination is $\mathcal{O}(n^3)$, banded systems with $k = \mathcal{O}(1)$: $\mathcal{O}(n)$, see tridiagonal matrix algorithm for $k = 1$ case.

Remark. Statement without proof: For arbitrary k we can solve k-diagonal matrix by running Gaussian elimination, where in the first phase we only zero out the elements below the main diagonal, and in the second phase only the elements above the diagonal. This way matrix will continue to be k-diagonal during transformations, so the whole process will work in $\mathcal{O}(nk^2) = \mathcal{O}(n)$, if implemented on sparse matrices.

General remarks:

- A nonsingular matrix is called *regular*, it has full rank and has an inverse.
- A square matrix is nonsingular if the determinant $\neq 0$.
- If the matrix is singular, GE (both versions) will lead to division by zero (zero pivots) at some stage, or a floating point exception. You don't have to check for this explicitely: once you get an exception, you know the matrix is singular.
- Errors. If we're trying to solve $Ax = b$ and we found the solution $x = \tilde{x}$, then the error is $r = A\tilde{x} - b$.

  If $\|r\|$ is large due to rounding, we call the matrix *ill-conditioned*.

3.4 LU Decomposition

- In many applications the same linear system has to be solved for different right hand sides.
- If you know the inverse, you can simply apply it to the right hand side:

$$x_1, x_2, x_3 \in \mathbb{R}^n, A \in \mathbb{R}^{n\times n}$$
$$Ax_1 = b_1,\ Ax_2 = b_2, Ax_3 = b_3$$
$$x_1 = A^{-1}b_1,\ x_2 = A^{-1}b_2,\ x_3 = A^{-1}b_3$$

- We want to store the operations of GE and backward substitution in a convenient way, so we can easily solve for different b.
- The key operation is: all the row operations of GE can be formulated as linear operators (which can be represented as matrices).

  In forward *elimination*, i.e. adding a multiple of an upper row to a lower row, the matrices are lower-triangular.

Example.

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

$M_1 A$ does the following:

- Leave the first row as is.
- Subtract 2 row 1 from row 2.
- Subtract $\frac{1}{2}$ row 1 from row 3.
- Adds row 1 to row 4.

Reducing echelon form in 3 steps by $M_3M_2M_1A$ and we get new system

$$Ax = b \Longleftrightarrow M_3M_2M_1A = M_3M_2M_1b, \text{ where } M_3M_2M_1 \text{ is an upper-triangular matrix}$$

Previous example gives us (GE without partial pivot):

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1/2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & 1/2 & 0 & 1 \end{bmatrix}, \quad M_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \end{bmatrix}$$

Remark.

- Product of lower triangular matrices is a lower triangular matrix.
- Inverse of a lower triangular matrix is a lower triangular matrix.

Summary: If $M_3M_2M_1A = U$, where U — upper-triangular matrix, then

$$A = (M_3M_2M_1)^{-1}U = M_1^{-1}M_2^{-1}M_3^{-1}U = LU$$

Let's formulate a theorem to somewhat formalize things.

Theorem 1. Let $A \in \mathbb{R}^{n\times n}$ be *invertible*. Then there exists a decomposition $LU = A$ with L — a lower triangular matrix, and U — upper triangular, $L, U \in \mathbb{R}^{n\times n}$ and $L = M_1^{-1}M_2^{-1}\dots M_{n-1}^{-1}$, where M_i is the matrix describing step i of forward elimination of GE. U is upper-triangular-matrix (echelon form).

$$U = M_{n-1}M_{n-2}\dots M_1 A$$

Remark. LU can be done with pivoting, called LUP decomposition. There will be other matricies involved which swap rows.

Example.

$$A = \begin{bmatrix} 6 & -2 & 2 & 4 \\ -12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix}$$

$$U = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

$$M_1^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} \text{ (notice the sign change)}$$

I.e. M_1^{-1} is an opposite operation to M_1: subtraction $\to$ addition, addition $\to$ subtraction.

$$\implies L = M_1^{-1}M_2^{-2}M_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1/2 & 3 & 1 & 0 \\ -1 & -1/2 & 2 & 1 \end{bmatrix}$$

Finally, for solving $Ax = b$ we see $Ax = b \iff LUx = b$. We do a substitution $y \coloneqq UX$. We solve for y first, which is easy, since L is triangular. Once we have y, we can solve $Ux = y$ for x via backward substitution.

Remarks:

- LU has same complexity as GE ($\mathcal{O}(n^3)$).
- Any of the two substitution steps are $\mathcal{O}(n^2)$.

3.5 Cholesky decomposition

***Definition* 6.** $B \in \mathbb{R}^{n\times n}$ is positive definite if and only if:

1. All eigenvalues are greater than 0.
2. $v \cdot Bv > 0$ when $v \neq 0$.
   $v \cdot Bv = 0 \iff v = 0$.

Statement 1. When B is symmetric and real, then all of the eigenvalues of it are real numbers.

***Definition* 7** (Cholesky decomposition)**.** When $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite, then A can be decomposed in the following form:

$$A = LDL^\intercal$$

where:

- L is lower triangular, $a_{ii} = 1$, $i = 1, \ldots, n$.
- D is diagonal matrix with positive entries
- Because D has only positive entries, we can we can take the "square root" of it, i.e. find $\tilde{D}$, such that $D = \tilde{D}\tilde{D}$. So,
  $$A = LDL^\intercal = (L\tilde{D})(\tilde{D}L^T) = \tilde{L}\tilde{L}^\intercal$$
  Here $\tilde{D} = D$ as D is diagonal.

With such a decomposition, we can solve $Ax = b$ as $Ax = \tilde{L}\tilde{L}^\intercal x = b$, where $\tilde{L}$ is lower-triangular and $\tilde{L}^\intercal$ is upper-triangular.

The advantage from LU decomp is that we only need to save one matrix, and forward substitution of the same kind will be carried out twice (so the matrix will already be in cache).

The complexity if $\mathcal{O}(n^3)$, but compared to LU approximately half the operations are needed.

How to compute $\widetilde{L}$? Let's suppose

$$\tilde{L} = \begin{bmatrix} l_{11} & 0 & \ldots & 0 \\ l_{21} & l_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \ddots \\ l_{n1} & \ldots & \ldots & l_{nn} \end{bmatrix}$$

Then

$$\begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & \ldots & 0 \\ \vdots & \ddots & \vdots \\ l_{n1} & \ldots & l_{nn} \end{bmatrix} \cdot \begin{bmatrix} l_{11} & \ldots & l_{n1} \\ \vdots & \ddots & \vdots \\ 0 & \ldots & l_{nn} \end{bmatrix}$$

Then it follows by the formula of matrix multiplications:

$$a_{ij} = \sum_{k=1}^{n} l_{ik} l_{jk}$$

Since a lot of entries in $\tilde{L}$ are zeros, actually

$$a_{ij} = \sum_{k=1}^{j} l_{ik} l_{jk}, \; i \geqslant j$$

We only write such equations for $i \geqslant j$, as we have a symmetrical matrix on the left hand side.

Now, for $i = j$:

$$a_{ii} = \sum_{k=1}^{i} l_{ik} l_{ik} = \sum_{k=1}^{i} (l_{ik})^2 \iff l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} (l_{ik})^2}$$

Here l_{ii} is non-negative, as in L the diagonal elements are 1, and $\tilde{L} = L \cdot \sqrt{D}$, which doesn't change the sign of the diagonal elements.

For $i > j$:

$$a_{ij} = \sum_{k=0}^{j} l_{ik} l_{jk}$$

and so:

$$l_{ij} = \frac{1}{l_{jj}} \Big(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \Big)$$

Finally $i < j : l_{ij} = 0$ as we have an lower triangular matrix.

This means we can compute entries of $\tilde{L}$ columnwise.

When does the algorithm fail? If we take the square root of a negative number at some point, that means that the matrix was non-positive-definite.

Algorithm:

For $i = 1 \dots n$:
  For $j = 1 \dots i - 1$ (here $i > j$)
    $y = a(i, j)$
    For $k = 1 \dots j - 1$:
      $y = y - l(i, k) \cdot l(j, k)$
    end for
    $l(i, j) = \frac{y}{l(j,j)}$
  end for
  $y = a(i, j)$ (here $i = j$)
  For $k = 1 \dots i - 1$:
    $y = y - l(i, k) \cdot l(i, k)$
  end for
  if $(y \leqslant 0)$ exit (no solution)
  else $l(i, j) = \sqrt{y}$
end for

The time complexity is $\mathcal{O}(n^3)$.

Example.

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{bmatrix}$$

Compute $\tilde{L}\tilde{L}^\intercal = A$. Columnwise:

$$l_{11} = \sqrt{a_{11}} = 1$$
$$l_{21} = \frac{1}{l_{11}}(a_{21} - 0) = \frac{1}{1} = 1$$
$$l_{31} = \frac{1}{l_{11}}(a_{31} - 0) = \frac{1}{1} = 1$$

Second column:

$$l_{22} = \sqrt{a_{22} - \sum_{k=1}^{n} l_{2k}^2} = \sqrt{a_{22} - l_{21}^2} = \sqrt{2-1} = 1$$
$$l_{32} = \frac{1}{l_{22}}(a_{32} - l_{31}l_{21}) = \frac{1}{1}(3-1) = 2$$

Third column:

$$l_{33} = \sqrt{a_{33} - \sum_{k=1}^{2} (l_{3k})^2} = \sqrt{a_{33} - (l_{31})^2 - (l_{32})^2} = \sqrt{6-5} = 1$$

So,

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Remark. For the exam: if you have the time, then check! Multiply the matrix by itself transposed, and see whether you got the original matrix back.

We can again solve the same equation for multiple ride hand sides:

- $Ax = b$ becomes $\tilde{L}\tilde{L}^\intercal x = b$.
- Solve $\tilde{L}y = b$.
- Solve $\tilde{L}^T x = y$.

4 Nonlinear equations

Linear equations can be represented by a matrix. On the other hand, nonlinear equations can't. Nonlinear equations are about finding roots. They appear in many applications.

Example. If we want to solve

$$x^2 + 3x + 7 = \log(x)$$

If we bring it all to one side, then we have to find the root of a function.

$$x^2 + 3x + 7 - \log(x) = 0$$

Unfortunately, non-linear equations are difficult to solve analytically. We can usually approximate the solution by using iterative methods.

4.1 Bisection method

Theorem 1. Let $f \in C([a,b])$ ($f(x)$ is continuous on $[a,b]$) with $f(a) \cdot f(b) < 0$ (i.e. f has different signs on its ends). Then, by continuity, there exists $r \in (a,b)$, such that $f(r) = 0$.

Remark. Actually, this theorem tells that any point between $f(a)$ and $f(b)$ will be achieved.

Idea of the bisection method:

1. Bisection of $[a,b]$ into $[a,c] \cup [c,b]$, i.e. into two subintervals, $a < c < b$.
2. If $f(c) = 0$, then $r = c$, we have found the solution.
3. If $f(c) \cdot f(a) < 0$, then continue with $[a,c]$.
4. If $f(c) \cdot f(b) < 0$, continue with $[c,b]$.

Remark. When $f(a) \cdot f(b) > 0$, f may or may not have a root on $[a,b]$ — we cannot say for sure.

Remark. This algorithm will only find one root, not all.

The method in short:

Producing a sequence of intervals $[a_i, b_i]$ such that a root r is inside these intervals. The starting interval is $[a,b] = [a_0, b_0]$.

Theorem 2. The bisection method, when applied to $[a,b]$ and $f \in C^*([a,b])$ with $f(a) \cdot f(b) < 0$ will complete after n steps an approximation c_n of root r with error

$$|r - c_n| < \frac{b-a}{2^n}$$

Remark. If we do infinitely many iterations, we will always find the root, because

$$\lim_{n\to\infty} \frac{b-a}{2^n} = 0$$

Proof. At every step, the length of the interval where the root is located is divided into two. We can use induction to prove the above-mentioned error formula. □

Example. Let $[a, b] = [0, 1]$. How many iterations are needed to decrease the error below $2^{-20} \approx 10^{-6}$?

We need to find an $n \in \mathbb{N}$, such that our error estimate is less than what we want:

$$\frac{b-a}{2^n} < 2^{-20} \iff \frac{b-a}{2^{-20}} < 2^n \iff \log_2\left(\frac{b-a}{2^{-20}}\right) \overset{\log_2 \text{ is strictly increasing}}{<} \log(2^n) = n$$

$$\log_2\left(\frac{1}{2^{-20}}\right) < n \iff \log_2(1) - \log_2(2^{-20}) < n \iff 0 - (-20) < n \iff 20 < n$$

Remark. The bisection method has a slow convergence. For every iteration step we get a *binary* digit as our accuracy increase. Recall that for approximating $\cos(x)$ by Taylor polynomial, we get 2 decimal digits per term of the polynomial.

4.2 Newton method

***Definition* 1.** Suppose that the sequence $\{x_n\}$ converges as $n \to \infty$. Then the sequence:

a) converges *linearly* to r, if there exists such an $M \in (0, 1)$, such that

$$\lim_{n\to\infty} \frac{|x_{n+1} - r|}{|x_n - r|} = M$$

Example. For bisection method $M = \frac{1}{2}$.

b) converges *super-linearly* to r, if

$$\lim_{n\to\infty} \frac{|x_{n+1} - r|}{|x_n - r|} = 0$$

c) converges *sub-linearly* to r, if

$$\lim_{n\to\infty} \frac{|x_{n+1} - r|}{|x_n - r|} = 1$$

(i.e. at some point the error decrease rate stagnates).

d) converges *with order* $q > 1$, if there exists $M > 0$, such that

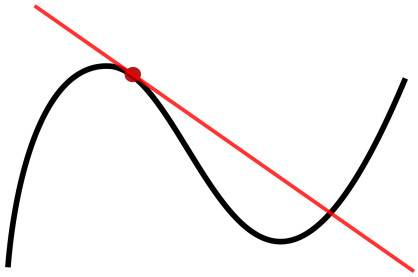
$$\lim_{n\to\infty} \frac{|x_{n+1} - r|}{|x_n - r|^q} < M$$

If $q = 2$, the convergence is called *quadratic*, if $q = 3$ — cubic.

***Definition* 2** (Newton method)**.** Let $f \in C^1([a, b])$. Then at every $x_0 \in (a, b)$ there exists a tangent to the graph of f. The tangent equation at x_0 can be written as follows:

$$t(x) = f(x_0) + f'(x_0)(x - x_0)$$
$$\frac{t(x) - f(x_0)}{x - x_0} = f'(x_0)$$

If we recall, the tangent at x_0 is the first order Taylor polynomial of $f(x)$ with $t(x_0) = f(x_0),\ t'(x) = f'(x_0)$.

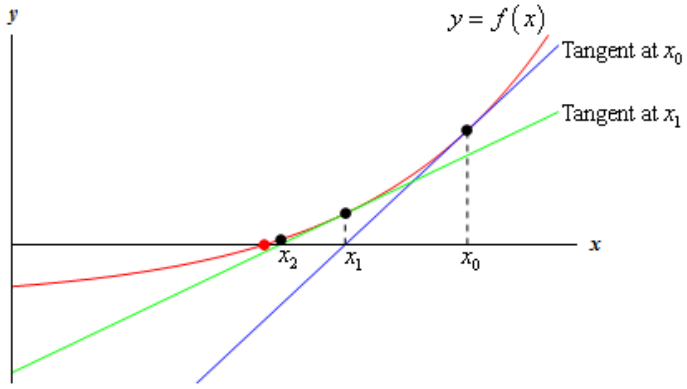


We use the tangent $t(x)$ as an approximation to $f(x)$, and we can find the root of $t(x)$ easily, as it's a linear funciton:

$$0 = t(x_1) = f(x_0) + f'(x_0)(x_1 - x_0) \iff x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Continuing yields the Newton sequence (Newton–Raphson iteration):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \ x_0 = \text{initial guess}$$



Here we cannot have $f'(x_n) = 0$, since we cannot divide by zero. Therefore, it's often a good idea to make sure f' doesn't have roots on the interval of search.

Theorem 3. Suppose we have a function $f \in C^2([a, b])$, such that:

1. $f(a) \cdot f(b) < 0$ (there's a sign switch).
2. f has no critical points in (a, b) (i.e. $f'(x) \neq 0$ for $x \in (a, b)$).
3. f'' exists, is continuous and either $f'' > 0$ or $f'' < 0$ on the whole (a, b). I.e., f is either concave down or concave up.

Then $f(x) = 0$ has *exactly one* solution r. The Newton sequence always converges to r.

For $n \to \infty$, when the initial guess should be chosen according to:

- If $f(a) < 0$, $f'' < 0$ or $f(a) > 0, f'' > 0$ then $x_0 \in [a, r]$, e.g. $x_0 = a$.
- If $f(a) < 0$, $f'' > 0$ or $f(a) > 0$, $f'' < 0$ then $x_0 \in [r, b]$, e.g. $x_0 = b$.

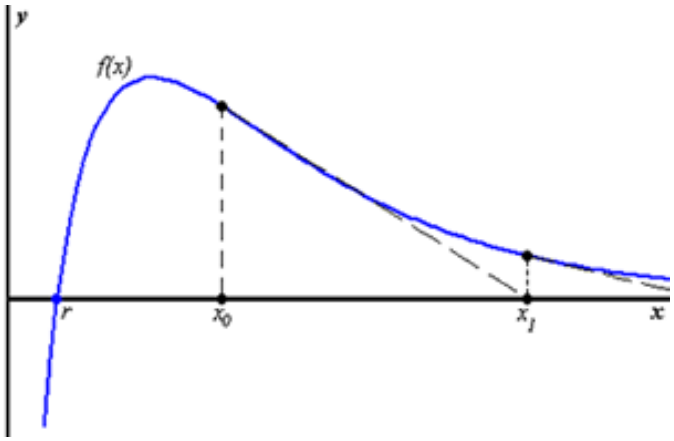
(If we chose an initial guess on the wrong side of the graph, then the Newton method may diverge outside the domain area of (a, b) — we don't want that.)

In any case we have the estimate

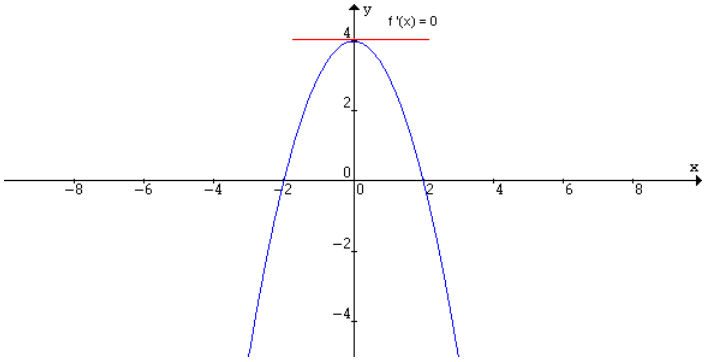
$$|x_n - r| < \frac{f(x_n)}{\min_{[a,b]} |f'(x)|}$$

Problems with the Newton method:

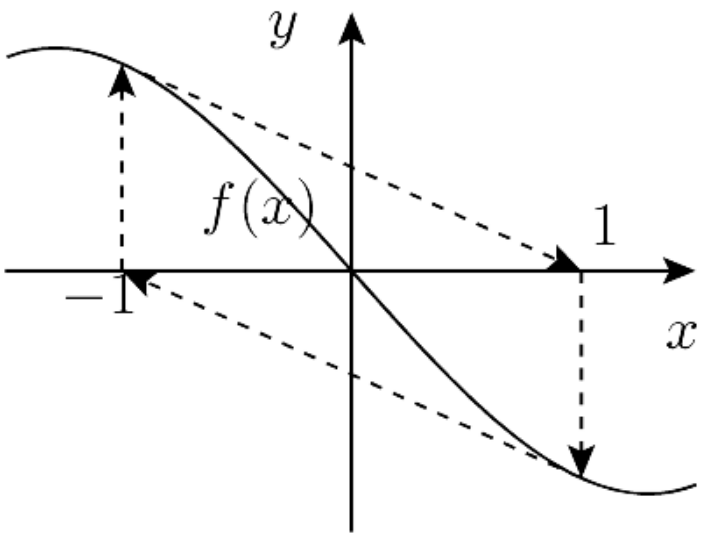
1. The runaway problem: the Newton method may start stepping in the wrong side.



2. If we have a flat point, we don't know in which direction to go:



3. Sometimes the Newton method can jump infinitely between two points in a cycle without ever converging.



Theorem 4. If $f : [a, b] \to \mathbb{R}$ fullfills:

1. $f(a) \cdot f(b) < 0$.
2. f has no critical points in (a, b).
3. f'' exists and is continuous ($f \in C^2([a, b])$) and x_0 is *"close enough"* (not a mathematical term) to the root r.

Then the Newton sequence converges quadratically.

Proof. We have to show that $|x_{n+1} - r| \leqslant M|x_n - r|^2$ as $n \to \infty$. By Taylor series approximation,

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(\xi_n)(x - x_n)^2, \text{ where } \xi_n \in (x, x_n)$$

Since r is a root of f, when we put $x = r$, we get

$$0 = f(r) = f(x_n) + f'(x_n)(r - x_n) + \frac{1}{2}f''(\xi_n)(r - x_n)^2$$

Let's divide both sides by $f'(x_n)$ and rearrange the terms:

$$\begin{aligned}
&\frac{f(x_n)}{f'(x_n)} + \frac{f'(x_n)}{f'(x_n)}(r - x_n) = -\frac{f''(\xi_n)}{2f'(x_n)}(r - x_n)^2 \\
&\iff (r - x_{n+1}) = -\frac{f''(\xi_n)}{2f'(x_n)}(r - x_n)^2 \quad \left(x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\right) \\
&\implies |r - x_{n+1}| = \left|\frac{f''(\xi_n)}{2f'(x_n)}\right| |r - x_n|^2
\end{aligned}$$

Let

$$M \coloneqq \sup_{x_n, \xi_n} \left|\frac{f''(\xi_n)}{2f'(x_n)}\right|$$

The supremum will be finite, because x_n and ξ_n are both inside bounded intervals.

Then $|r - x_{n+1}| \leqslant M|r - x_n|^2$. □

Remark. We need x_0 to be close enough to make sure the Newton method doesn't go outside the domain $[a, b]$.

Modified Newton method

As we remember, in the usual Newton method

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

If $f'(x_n)$ doesn't change much anymore from n to $n + 1$, we can fix it to $f'(x_m)$ with some fixed $m \leqslant n$. Then our new formula for x_{n+1} is

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

We avoid futher evaluations of $f'(x)$ which saves computational resources.

4.3 Secant method

Newton's method needs derivatives. We can approximate the derivatives by secants (also called a difference quotient):

$$f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} \implies f'(x) \approx \frac{f(x+h)-f(x)}{h} \text{ for small } h$$

For the Secant method, we let $x_n \coloneqq x_n,\ h \coloneqq x_{n-1} - x_n$.

$$f'(x_n) \approx \frac{f(x_{n-1})-f(x_n)}{x_{n-1}-x_n}$$

Substitution into Newton method gives us

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

The Secant method converges slower than the Newton method, but it's easier to compute.

Theorem 5. If $f \in C^2([a,b])$ and $r \in (a,b),\ f(r)=0,\ f'(r)\neq 0$ and

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

Then there exists a $\delta > 0$, such that for every x_1, x_0, such that $|r-x_1|<\delta$, $|r-x_0|<\delta$, we'll have

1. $\lim_{n\to\infty}|x_n - r| = 0$ (the sequence converges)
2. $|r-x_{n+1}| \leqslant M|r-x_n|^\alpha$ with $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$ (the sequence converges with order of golden ratio).

Example.

$$f(x) = x^5 - 3x + 1 = 0,\ f'(x) = 5x^4 + 3$$

Newton method:

$$x_0 = 0,\ f(0) = 1,\ f'(0) = 3$$
$$x_1 = x_0 = \frac{f(x_0)}{f'(x_0)} = 0 - \frac{1}{3} = -\frac{1}{3}$$

Secant method:

$$x_0 = 1,\ x_1 = \frac{1}{2}$$
$$x_2 = \frac{1}{2} - \frac{f(1/2)(1-1/2)}{f(1)-f(1/2)} = \dots$$

Summary:

	#initials	Regularity of f	Convergence	Root between points	Function calls per iteration
Bisection	2 (a, b)	$C([a,b])$	linear	Yes	1
Newton	1 (x_0)	$C^2([a,b])$	quadratic	No	2
Secant	2 (x_0, x_1)	$C^2([a,b])$	superlinear/order $\alpha = 1.618$	No	1

4.4 Systems of non-linear equations

Given a *vector-valued* function $f : \mathbb{R}^m \to \mathbb{R}^m$ we want to solve the equation $f(\vec{x}) = \vec{0}$, i.e. find a vector $\vec{r} \in \mathbb{R}^m$, such that $f(\vec{r}) = \vec{0}$. Such a function f is represented by

$$f(\vec{x}) = \begin{pmatrix} f_1(\vec{x}) \\ \vdots \\ f_m(\vec{x}) \end{pmatrix}$$

where $f_i : \mathbb{R}^m \to \mathbb{R}$ for $i = 1, \dots m$ and vectors $\vec{x} = (x_{(1)}, \dots, x_{(m)}) \in \mathbb{R}^m$.

Thus, $f(\vec{x}) = \vec{0}$ means that we need to simultaneously solve $f_i(\vec{x}) = 0$ for all $i = 1, \dots, m$.

The generalization of derivatives for such functions is the so called *Jacobian matrix*:

$$Df = \begin{bmatrix} \frac{\partial f_1}{\partial x_{(1)}} & \frac{\partial f_1}{\partial x_{(2)}} & \cdots & \frac{\partial f_1}{\partial x_{(m)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_{(1)}} & \cdots & \cdots & \frac{\partial f_m}{\partial x_{(m)}} \end{bmatrix} \in \mathbb{R}^{m \times m}$$

where $\frac{\partial f_i}{\partial x_{(j)}}$ is a partial derivative, which tells you the slope of the function f_i in the direction x_i.

Example. Let $m = 3$, i.e. $f : \mathbb{R}^3 \to \mathbb{R}^3$. Let

$$f_1(\vec{x}) = \begin{pmatrix} x_{(1)} + x_{(2)} + x_{(3)} - 3 \\ x_{(1)}^2 + x_{(2)} + x_{(3)}^2 - 5 \\ e^{x_{(1)}} - x_{(1)}x_{(2)} - x_{(1)}x_{(3)} - 1 \end{pmatrix}$$

Then the Jacobi/Jacobian matrix is

$$(Df)(\vec{x}) = \begin{bmatrix} 1 & 1 & 1 \\ 2x_{(1)} & 1 & 2x_{(3)} \\ e^{x_{(1)}} - x_2 - x_3 & -x_{(1)} & -x_{(1)} \end{bmatrix}$$

Newton's method can be generalized to this setting:

$$\vec{x}_{n+1} = \vec{x}_n - (Df)^{-1}(\vec{x}_n) f(\vec{x}_n)$$

In practice, the inverse $(Df)^{-1}$ need not to be calculated, but a system

$$\big(Df(\vec{x}_n)\big)\vec{y} = f(\vec{x}_n)$$

may be solved. Then

$$\vec{y} = \big(Df(\vec{x}_n)\big)^{-1} f(\vec{x}_n)$$

We need the Jacobian to be nonsingular at the root. As derivatives are required to be continuous, Jacobian will be nonsingular in some vicinity of the root.

5 Interpolation and approximation

We have some measurements under certain conditions. Now we want to estimate what happens in situations we have not measured (yet).

Example. Viscosity of water measured for various temperatures:

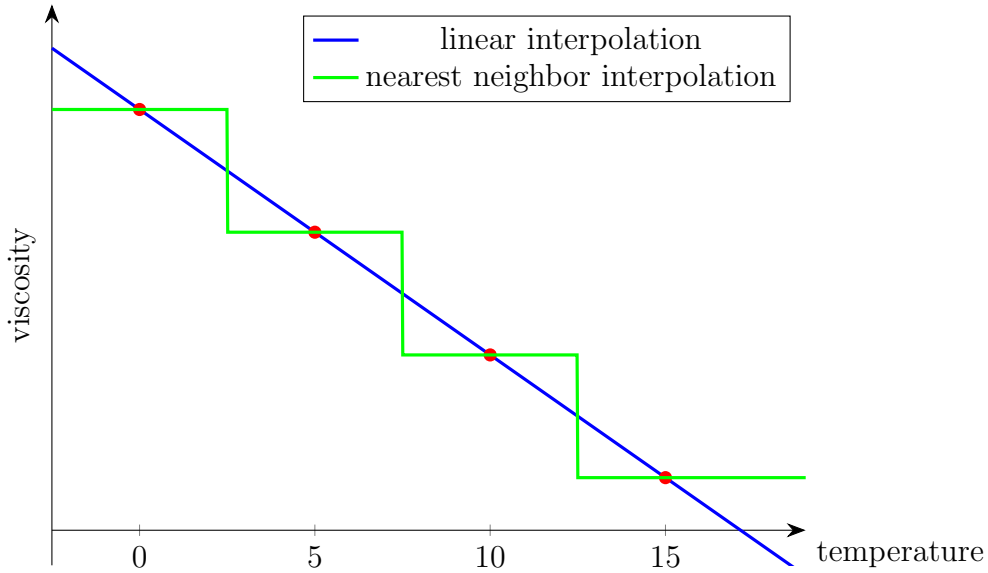
temperature/°C	0	5	10	15
viscosity/cP	1.792	1.519	1.308	1.140

$$1 \text{ P (Poise)} = 0.1 \frac{N}{m^2 \cdot s^{-1}}$$

where $\frac{N}{m^2}$ — force per area, s^{-1} — rate of shear, the whole thing measures resistance towards shear.

Question: what is the viscosity at temperature = 8 °C?

Solution:



Find a *linear* function $v(t)$ that achieves the measured values at $T = 5$ and $T = 10$. Then evaluate this function at $T = 8$.

$$v(T) = \frac{T-5}{10-5}T_{10} + \frac{10-T}{10-5}T_5 = \frac{T-5}{5}T_{10} + \frac{10-T}{5}T_5$$

and so

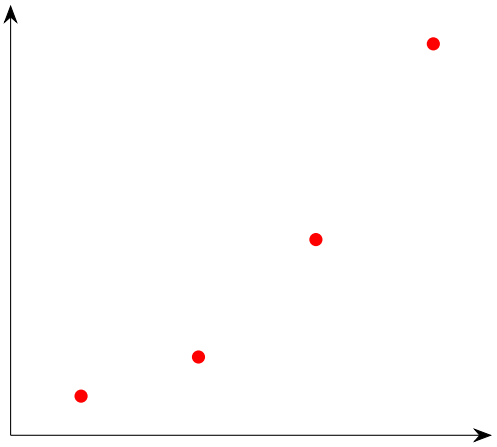
$$v(8) = \frac{3}{5}T_{10} + \frac{2}{5}T_5$$

Note that here $\frac{3}{5}$ and $\frac{2}{5}$ are ratios of how 8 divides $[5, 10]$.

That means, for linear interpolation, T_{10} and T_5 need to be weighted with the cut-ratios $\frac{2}{5}$, $\frac{3}{5}$ abd added together ($T_5 = 1.519,\ T_{10} = 1.308$).

5.1 Polynomial interpolation

What if the measured values look like this?



We can either:

1. Approximate by piecewise linear interpolation.
2. Use nonlinear interpolation.

***Definition* 1** (Interpolating function)**.** Given values $p_0, \dots, p_n \in \mathbb{R}$ and nodes $u_0, \dots, u_n \in \mathbb{R}$ a function $p : \mathbb{R} \to \mathbb{R}$ with $p(u_i) = p_i,\ i = 1, \dots, n$ is called *interpolating*.

Remark. This definition can be generalized to points $p_i \in \mathbb{R}^d$ and curves $p : \mathbb{R} \to \mathbb{R}^d$.

The goal of polynomial interpolation: to find coefficients $\alpha_0, \dots, \alpha_n \in \mathbb{R}$, such that for a given set of polynomials $\varphi_0, \dots, \varphi_n$ the linear combination

$$p(u) \coloneqq \sum_{i=0}^{n} \alpha_i \varphi_i(u)$$

is interpolating. This means

$$p(u_j) = \sum_{i=0}^{n} \alpha_i \varphi_i(u_j) = p_j,\ j = 0, \dots, n$$

In other words, $\alpha_0, \dots, \alpha_n$ shall be such that

$$\begin{cases} \sum_{i=0}^{n} \alpha_i \varphi_i(u_0) = p_0 \\ \vdots \\ \sum_{i=0}^{n} \alpha_i \varphi_i(u_n) = p_n \end{cases}$$

This is a system of linear equations for weights α_i:

$$\Phi \vec{\alpha} = \vec{p}$$
$$\vec{\alpha} = (\alpha_0, \dots, \alpha_n),\ \vec{p} = (p_0, \dots, p_n) \in \mathbb{R}^{n+1}$$
$$\Phi \in \mathbb{R}^{(n+1)\times(n+1)}$$
$$\Phi = \begin{bmatrix} \varphi_0(u_0) & \dots & \varphi_n(u_0) \\ \vdots & \ddots & \vdots \\ \varphi_0(u_n) & \dots & \varphi_n(u_n) \end{bmatrix} \text{ is called } \textit{collocation matrix}$$

It follows that $\vec{\alpha} = \Phi^{-1}\vec{p}$.

Φ is invertible if and only if the set of functions $\varphi_0, \ldots, \varphi_n$ is linearly independent $\iff$ they are a basis of the respective interpolation space, here a basis of polynomials of degree $\leqslant n$.

Example. Let p be a polynomial of degree $\leqslant n$. Let $\varphi_i(x) = x^i$, i.e. $\varphi_0(x) = 1$, $\varphi_1(x) = x$, and so on.

$$x^2 + 2x + 3 = \varphi_2(x) + 2\varphi_1(x) + 3\varphi_0(x)$$

Let $u_i = i + 1,\ i = 0, \ldots, n$. Collocation matrix:

$$\Phi = \begin{bmatrix} \varphi_0(u_0) & \ldots & \varphi_n(u_0) \\ \vdots & \ddots & \vdots \\ \varphi_0(u_n) & \ldots & \varphi_n(u_n) \end{bmatrix} = \begin{bmatrix} 1^0 & 1^1 & \ldots & 1^n \\ 2^0 & 2^1 & \ldots & 2^n \\ \vdots & \vdots & \ddots & \vdots \\ (n+1)^0 & (n+1)^1 & \ldots & (n+1)^n \end{bmatrix} \text{ — Vandermonde matrix}$$

As we can see, a Vandermonde matrix has very small entries (1) and very big ones ($(n+1)^n$), therefore rounding errors can become relatively large even for small n. Therefore, $\varphi_i(x) = x^i$ is a bad choice of basis functions.

Consider $n = 2$ and values p_i as

u_i	1	2	3
p_i	2	5	10

Then

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} -2 \\ 5 \\ 10 \end{pmatrix}$$

(basis functions at nodes $u_i \cdot$ coefficients = values)

$$\begin{aligned} \alpha_0 = 1,\ \alpha_1 = 0,\ \alpha_2 = 1 \implies p(u) &= 1 \cdot \varphi_0(u) + 0 \cdot \varphi_1(u) + 1 \cdot \varphi_2(u) \\ &= 1 \cdot 1 + 0 \cdot u + 1 \cdot u^2 \\ &= 1 + u^2 \text{ — the interpolating polynomial} \end{aligned}$$

Can we find bases for polynomial spaces that leads to a more convenient collocation matrix?

5.1.1 Lagrange interpolation

Given $n + 1$ values/points $p_1, \ldots, p_n$ and corresponding nodes $u_0, \ldots, u_n$ define the interpolating polynomial as

$$p(u) = \sum_{i=0}^{n} p_i L_i^n(u) \quad (n \text{ here is an upper index, not power})$$

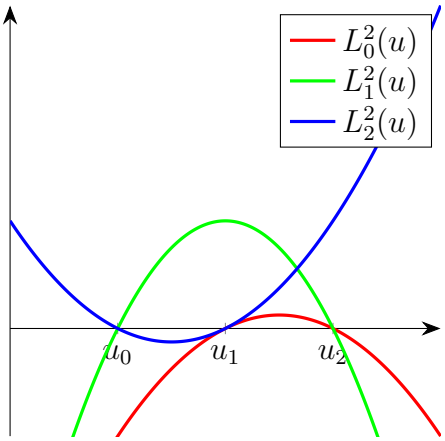
where the so called Lagrange polynomial $L_i^n(n)$ of degree n fullfills:

$$L_i^n(u_j) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

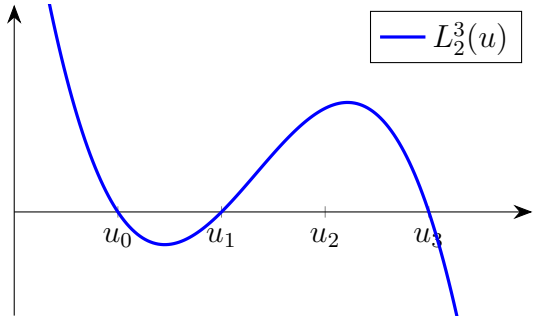
(Such a function δ is called Kronecker delta.)

Example.

- Lagrange polynomials of degree 2:



- Degree 3:



For $n = 2$ these polynomials are defined as

$$L_0^2(u) = \frac{(u-u_1)(u-u_2)}{(u_0-u_1)(u_0-u_2)}$$
$$L_1^2(u) = \frac{(u-u_0)(u-u_2)}{(u_1-u_0)(u_1-u_2)}$$

The numerator makes sure that the polynomial is equal to zero at other nodes, and the denominator scales the whole polynomial so that it is equal to 1 at u_1.

In general, for $n \in \mathbb{N}$,

$$L_i^n(u) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(u-u_j)}{(u_i-u_j)}$$

By construction, $L_i^n(u_j) = \delta_{ij}$.

Example.

u_i	0	1	2
p_i	2	4	3

Step 1. Get Lagrange polynomial:

$$L_0^2(u) = \frac{(u-u_1)(u-u_2)}{(u_0-u_1)(u_0-u_2)} = \frac{(u-1)(u-2)}{(-1)\cdot(-2)} = \frac{1}{2}u^2 - \frac{3}{2}u + 1$$
$$L_1^2(u) = \cdots = -u^2 + 2u$$
$$L_2^2(u) = \cdots = \frac{1}{2}u^2 - \frac{1}{2}u$$

Step 2. Collocation matrix:

$$\Phi = \begin{bmatrix} L_0^2(u_0) & L_1^2(u_0) & L_2^2(u_0) \\ L_0^2(u_1) & L_1^2(u_1) & L_2^2(u_1) \\ L_0^2(u_2) & L_1^2(u_2) & L_2^2(u_2) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I$$

Step 3. $\Phi\vec{\alpha} = \vec{p} \iff \vec{\alpha} = \vec{p}$ so $p(u) = 2L_0^2(u) + 4L_1^2(u) + 3L_2^2(u)$

Consequence. In Lagrange interpolation, the collocation matrix (by construction) is the identitiy matrix!

Summary:

- In Lagrange interpolation, basis functions are such that interpolation scheme becomes trivial.
- However, if one more node is added, the computations have to be redone.

5.1.2 Aitken's algorithm

We want to evaluate $p(u)$ at one location u^*. Then we can use Aitken's algorithm without directly computing $p(u)$.

$$L_i^n(u) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(u - u_j)}{(u_i - u_j)}$$

The Lagrange polynomials have a so-called *partition of unity* property:

$$\sum_{i=0}^{n} L_i^n(u) = 1$$

(for proof use, e.g., $f(u) = 1$ as function that needs to be interpolated).

This allows for recursive definition of the Lagrange polynomials:

- For $n = 0$: $L_0^0(u) = 1$.
- For $n > 0$:

$$L_i^n(u) = L_i^{n-1}(u) \cdot \frac{u - u_n}{u_1 - u_n} \text{ for } i = 0, \ldots, n-1$$

(The Lagrange polynomials from previous n get one more root.)

And

$$L_n^n(u) \overset{\text{partition of unity}}{=} 1 - \sum_{i=0}^{n-1} L_i^n(u) = 1 - \sum_{i=0}^{n-1} L_i^{n-1}(u)\Big(\frac{u - u_n}{u_i - u_n}\Big) \overset{\text{partition of unity}}{=}$$

$$= \sum_{i=0}^{n-1} L_i^{n-1}(u) - \sum_{i=0}^{n-1} L_i^{n-1}(u) \cdot \Big(\frac{u - u_n}{u_i - u_n}\Big) = \sum_{i=0}^{n-1} L_i^{n-1}(u)\Big(1 - \frac{u - u_n}{u_i - u_n}\Big)$$

This recursive formula is basis for a simple algorithm to evaluate the Lagrange interpolating polynomials (Aitken's algorithm).

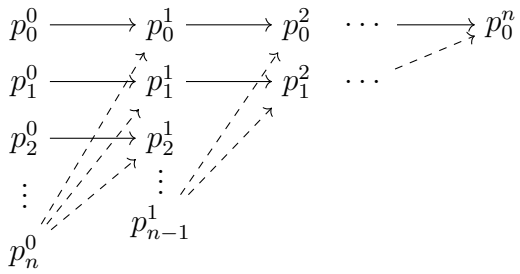
$$
\begin{aligned}
p(u) &= \sum_{i=0}^{n} p_i^0 L_i^n(u), \text{ where } p_i^0 = p_i \text{ — given data} \\
&= \sum_{i=0}^{n-1} p_i^0 L_i^n(u) + p_n^0 L_n^n(u) \\
&= \sum_{i=0}^{n-1} p_i^0 L_i^{n-1}(u)\frac{u-u_n}{u_i-u_n} + p_n^0 \sum_{i=0}^{n-1} L_i^{n-1}(u)\left(1-\frac{u-u_n}{u_i-u_n}\right) \text{ (from the recursive formula)} \\
&= \sum_{i=0}^{n-1} L_i^{n-1}(u)\left(p_i^0\frac{u-u_n}{u_i-u_n} + p_n^0\left(1-\frac{u-u_n}{u_i-u_n}\right)\right) =: p_i^1 \text{ for } i=0,\dots,n-1 \\
&= \sum_{i=0}^{n-1} p_i^1 L_i^{n-1}(u) = \cdots = \sum_{i=0}^{n-2} p_i^2 L_i^{n-2}(u), \text{ where } p_i^2 := p_i^1\frac{u-u_{n-1}}{u_i-u_{n-1}} + p_{n-1}^1\left(1-\frac{u-u_{n-1}}{u_i-u_{n-1}}\right) \\
&= \cdots = \sum_{i=0}^{0} p_i^n L_i^0(u) = p_0^n
\end{aligned}
$$

and

$$p_i^{k+1} = p_i^k\frac{u-u_{n-k}}{u_i-u_{n-k}} + p_{n-k}^k\left(1-\frac{u-u_{n-k}}{u_i-u_{n-k}}\right)$$

Evaluating p_0^n for a fixed value of n is equivalent to computing $p(u)$!

Scheme:



where $\longrightarrow$ means:

$$\cdot\,\frac{u-u_{n-k}}{u_i-u_{n-k}}$$

and $\dashrightarrow$ means:

$$\cdot\left(1-\frac{u-u_{n-k}}{u_i-u_{n-k}}\right)$$

Example.

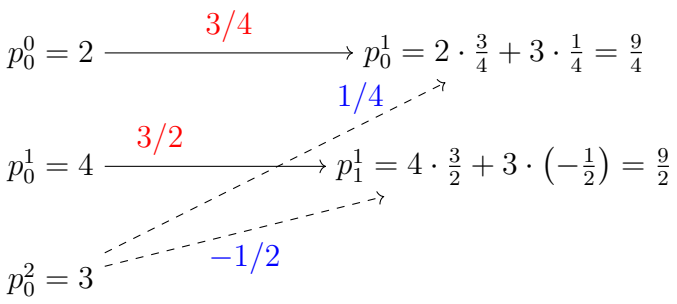
u_i	0	1	2
p_i	2	4	3

We want to compute $p(1/2)$. $n=2$. For $k=0$ we have $p_0^0=2$, $p_1^0=4$, $p_2^0=3$. We need to calculate the coefficients to get from p_i^0 to p_i^1:

$$\text{for } i=0: \quad \frac{u-u_2}{u_0-u_2} = \frac{\frac{1}{2}-2}{0-2} = \frac{3}{4} \qquad 1-\frac{3}{4}=\frac{1}{4}$$

$$\text{for } i=1: \quad \frac{u-u_2}{u_1-u_2} = \frac{\frac{1}{2}-2}{1-2} = \frac{3}{2} \qquad 1-\frac{3}{2}=-\frac{1}{2}$$

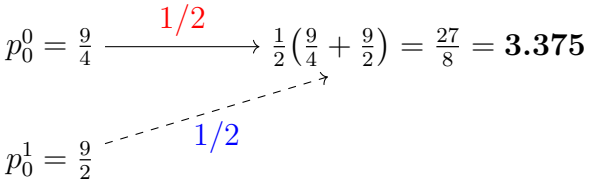
Now compute p_i^1:



For $k = 1$, we have $p_0^1 = \frac{9}{4}$, $p_1^1 = \frac{9}{2}$. For $i = 0$:

$$\frac{u - u_1}{u_0 - u_1} = \frac{\frac{1}{2} - 1}{0 - 1} = \frac{1}{2} \qquad 1 - \frac{1}{2} = \frac{1}{2}$$

Now compute p_0^2:



Summary: Aitken's algorithm is an iterative process for evaluating Lagrange interpolation polynomials without actually constructing them. It has a complexity of $\mathcal{O}(n^2)$, but it needs just half of the steps that are needed by explicit construction of $p(n)$.