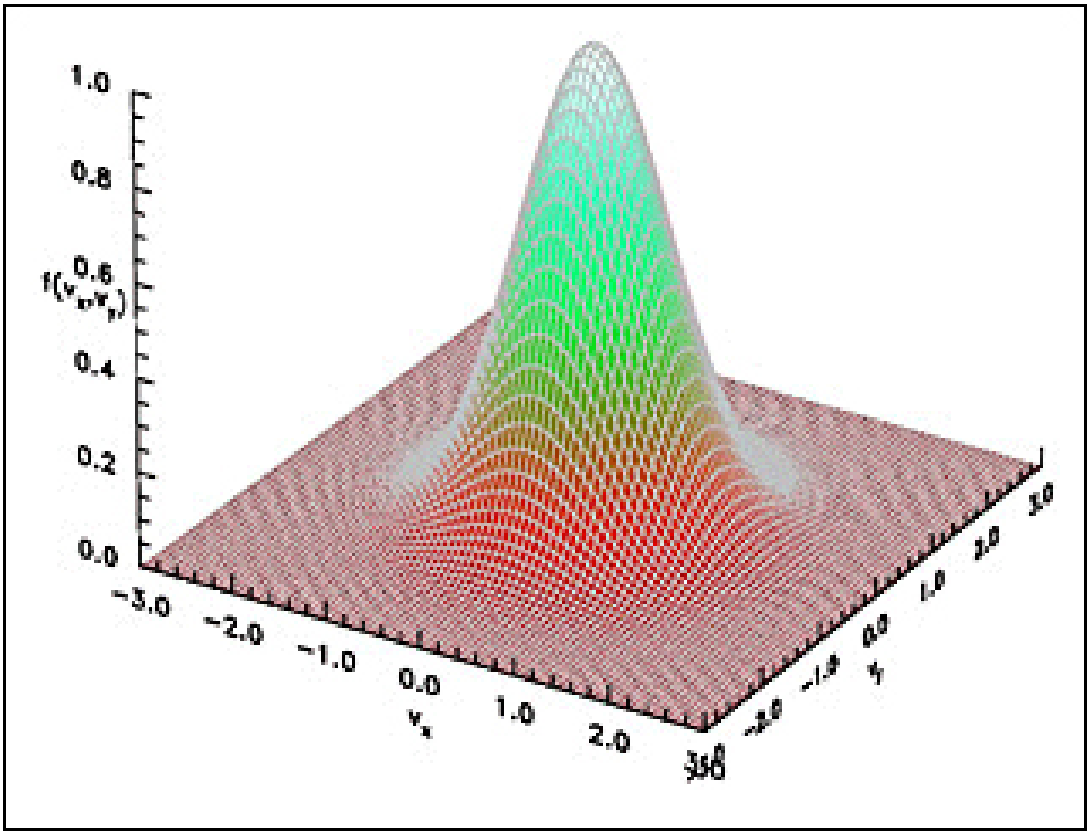
CSIE5123: Numerical Method

Vinsong

June 8, 2026

Abstract

The lecture note of 2026 Spring Numerical Method by professor 林智仁.



Contents

Chapter 1

Floating-point systems

Lecture 1

2025.12.09

1.1 Floating-point basics

This chapter is mainly based on the science of floating-point arithmetics which is based on the IEEE standard 754.

> **Remark.** The floating-point number system here introduction is to help us redesign a better floating-point system while we are doing deep learning implementation.

1.1.1 Why learning floating-point operations?

> **Example.** A one-variable problem
>
> $$\min_x f(x) \quad \text{where } x \geq 0$$

In the normal program, we should set an upper bound of x, or x may be wrongly increased to ∞. We have to find the largest representable number in the computer. Or how to define the "infinity" in the computer?

> **Example.** A ten-variable problem
>
> $$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{where } x_i \geq 0, i = 1, 2, \ldots, 10$$

We want to know how many are zeros, we may use

```
for (int i = 0; i < 10; i++)
    if (x[i] == 0) count++;
    // count the number of zeros
```

But people say that don't do the comparison of floating-point due to the precision issue.

```
double epsilon = 1.0e-12;
for (int i = 0; i < 10; i++)
    if (x[i] <= epsilon) count++;
```

Which is better? How to chose "`epsilon`"? Can't do the comparison of floating-point? We need to understand the floating-point representation.

1.1.2 Floating-point Format

We know `float` (single precision): 4 bytes and `double` (double precision) 8 bytes in C/C++.

Note. We usually spare the bits for deeplearning system to save memory and speed up the training process.

Definition 1.1.1 (format). A floating-point system requires

- a base β
- precision p
- significand (mantissa) $d.d\dots d$

Example.

$$0.1 = 1.00 \times 10^{-1} \qquad (\beta = 10, p = 3)$$
$$\approx 1.1001 \times 2^{-4} \qquad (\beta = 2, p = 5)$$

exponent: -1 and -4; significand: 1.00 and 1.1001.

We let $e_{\max}$ to be the largest exponent and $e_{\min}$ to be the smallest exponent. Then, β^p is the possible significand, and $e_{\max} - e_{\min} + 1$ is the possible exponent.

$$\lceil \log_2(e_{\max} - e_{\min} + 1) \rceil + \lceil \log_2(\beta^p) \rceil + 1$$

bits for storing a floating-point number. 1 bit for the sign.

In the pratical it is more complicated.

Definition 1.1.2 (Normalized floating-point number). A normalized floating-point number is a number whose significand is in the form of $1.d_1 d_2 \dots d_{p-1}$.

Example. 0.1 is not; 1.00×2^{-3} is

Now the issue will become represent of zero. We natually use $1.0 \times \beta^{e_{\min}-1}$ to represent zero, but it is not allow in computer system.

Remark. We usually reserve some special numbers to represent 0, ∞, and NaN.

1.1.3 Relative error and Ulps

Example. When $\beta = 10, p = 3$, for 3.14159

We represent it as 3.14×10^0, and the error is $|3.14159 - 3.14| = 0.00159 = 0.159 \times 10^{-2}$ i.e.

- 10^{-2} is the unit of the last place.
- `ulps`: 0.159 is the unit in the last place

The absolute error is 0.159 ulps.

Definition 1.1.3 (ulps). The unit in the last place (ulps) is the unit value of the least significant digit in a floating-point number.

We also use "relative error" to measure the error. The relative error is $0.00159/3.14159 \approx 0.0005$.
For a number $d.d\ldots d \times \beta^e$, the largest error cause by rounding is

$$0.\underbrace{0\ldots0}_{p-1}\beta' \times \beta^e \quad \beta' = \beta/2$$

So, Error $= \frac{\beta}{2} \times \beta^{-p} \times \beta^e$ and

$$1 \times \beta^e \leq \text{original number} < \beta \times \beta^e$$

The relative error will between

$$\frac{\frac{\beta}{2} \times \beta^{-p} \times \beta^e}{\beta^e} \text{ and } \frac{\frac{\beta}{2} \times \beta^{-p} \times \beta^e}{\beta^{e+1}}$$

So,

$$\text{relative error} \leq \frac{\beta}{2} \times \beta^{-p} \tag{1.1.1}$$

Definition 1.1.4 (machine epsilon). The machine epsilon is the upper bound of the relative error of a floating-point number $\epsilon = \frac{\beta}{2} \times \beta^{-p} = \beta^{1-p}/2$

Example. $x = 12.35 \Rightarrow \tilde{x} = 1.24 \times 10^1 \quad (p = 3,\ \beta = 10)$

- 1 ulps $= 0.01 \times 10^1 = 0.1$, $\epsilon = 1/2 \cdot 10^{-2} = 0.005$
- error $= 0.05 = 0.005 \times 10^1 = 0.5$ ulps
- relative error $= 0.05/12.35 \approx 0.004 = 0.8\epsilon$
- $8x = 98.8,\ 8\tilde{x} = 9.92 \times 10^1$ error $= 4.0$ ulps, relative error $= 0.04 = 0.8\epsilon$

Note. ulps and ϵ is used interchangeably usually.

1.2 Guard digits

In some situation, the order of floating-point computation and rounding may not have a huge effect on the final result. So, usually we choose to

$$\boxed{\text{Round}} \quad \Rightarrow \quad \boxed{\text{Compute}}$$

to let the computation to be more efficient. But in some cases, the error may be huge.

Example.

$$x = 10.1$$
$$y = 9.93$$
$$x - y = 0.17$$

- Round and then compute:

$$10.1 - 9.93 = 1.01 \times 10^1 - 0.99 \times 10^1 = 0.02 \times 10^1 = 2.00 \times 10^{-1}$$

  - error $= 2.00 \times 10^{-1} - 0.17 = 0.03$
  - ulps $= 0.01 \times 10^{-1} = 10^{-3}$
  - error $= 30$ ulps
  - relative error

$$= \frac{0.03}{0.17} = \frac{3}{17}$$

- Compute and then round:

$$10.1 - 9.93 = 0.17 \approx 1.7 \times 10^{-1}$$

  - error $= 1.7 \times 10^{-1} - 0.17 = 0$

So, we have to know how big will the error be if we choose to round first then compute.

Theorem 1.2.1. Using p precision with base β for $x - y$, the relative error can be as large as $\beta - 1$

Proof. Let

$$x = 1.0\ldots0,\ y = 0.\underbrace{\eta\ldots\eta}_{p \text{ digits}},\quad \eta = \beta - 1$$

The correct solution is $x - y = \beta^{-p}$. Under some rounding scheme, we have the solution

$$1.0\ldots0 - 0.\underbrace{\eta\ldots\eta}_{p-1 \text{ digits}} = \beta^{-p+1}$$

> **Note.** Above is a bad rounding just ignore the last η

Relative error is

$$\frac{|\beta^{-p+1} - \beta^{-p}|}{\beta^{-p}} = \beta - 1$$

■

Example. ($p = 3$, $\beta = 10$) $x = 1.00,\ y = 0.999,\ x - y = 0.001 = 10^{-3}$

The compute solution is $1.00 \times 10^0 - 0.99 \times 10^0 = 0.01 \times 10^0 = 0.01$. The relative error is

$$\frac{|0.01 - 0.001|}{0.001} = 9$$

Such a huge error occurs when the two numbers are close to each other. So, we need to use guard digits to store the intermediate result to avoid such a huge error.

Definition 1.2.1 (guard digits). Guard digits are extra digits used in the intermediate steps of a calculation to reduce the effect of rounding errors and improve the accuracy of the final result.

In here, the guard digits is p increased by 1 in the device for addition and subtraction

For the above example, we have the computation become

$$1.010 \times 10^1 - 0.993 \times 10^1 = 0.017 \times 10^1 = 0.17$$

note that $0.17 = 1.70 \times 10^{-1}$ can be stored in the system with $p = 3$ and $\beta = 10$. The relative error becomes 0.

Note. We only allocate more digit precision for the calculation device (i.e. the substraction process) to store the intermediate result.

Example. ($p = 3$, $\beta = 10$) $x = 110,\ y = 8.59, x - y = 101.41$

$$\begin{aligned} 110 - 8.59 &= 1.100 \times 10^2 - 0.085 \times 10^2 \\ &= 1.015 \times 10^2 \approx 1.01 \times 10^2 \end{aligned}$$

Relative error is

$$\frac{|1.01 \times 10^2 - 101.41|}{101.41} \approx 0.003$$

$\epsilon = \frac{1}{2} \cdot 10^{-2} = 0.005$.

Theorem 1.2.2. Using $p+1$ digits for $x - y \ \Rightarrow$ relative rounding error $< 2\epsilon$ (ϵ: machine epsilon)

Proof. We first make some assumption

- $x > y$
- $x = x_0 \,.\, x_1 x_2 \ldots x_{p-1} \times \beta^0$, the proof will be similar for not β^0

For the first two condition,

1. If $y = y_0 \,.\, y_1 y_2 \ldots y_{p-1}$ no error
2. If $y = 0.y_1 y_2 \ldots y_p \ \Rightarrow$ 1 guard digit, exact $x - y$ rounded to a closest number $\Rightarrow$ error $< \epsilon$

In general, $y = 0.0 \ldots 0\ y_{k+1} \ldots y_{k+p}$, we let $\bar{y}$: truncate y to $p+1$ digits.

$$|y - \bar{y}| < (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \cdots + \beta^{-p-k}) \tag{1.2.1}$$

After the truncation, we have to compute

$$x - \bar{y}$$

Then we rounded the result get

$$x - \bar{y} + \delta$$

Thus

$$\text{error} \le 0.\ \underbrace{0 \ldots 0}_{p-1 \text{ digits}}\ (\beta/2) \quad \text{(worst case)}$$

Therefore,

$$|\delta| \le (\beta/2) \cdot \beta^{-p} \tag{1.2.2}$$

Then the error is

$$(x - y) - (x - \bar{y} + \delta) = \bar{y} - y - \delta$$

Now, we consider three cases:

1° Case 1: $x - y \geq 1$, from (1.2.1) and (1.2.2), we have the relative error

$$\begin{aligned}\frac{|\bar{y} - y - \delta|}{x - y} &\leq \frac{|\bar{y} - y - \delta|}{1} \\ &\leq \beta^{-p}[(\beta - 1)(\beta^{-1} + \beta^{-2} + \cdots + \beta^{-k}) + \beta/2] \\ &= \beta^{-p}[(\beta - 1)\beta^{-k}(1 + \cdots + \beta^{k-1}) + \beta/2] \\ &= \beta^{-p}[(\beta - 1)\beta^{-k} \cdot \frac{\beta^k - 1}{\beta - 1} + \beta/2] \\ &= \beta^{-p}[(1 - \beta^{-k}) + \beta/2] \\ &< \beta^{-p}[1 + \beta/2] \leq 2\epsilon\end{aligned}$$

2° Case 2: $x - \bar{y} \leq 1$, enough digits to store $x - \bar{y}$, so the $\delta = 0$, and the relative error is

$$\frac{|\bar{y} - y|}{x - y}$$

The smallest $x - y$ is

$$1.0 - 0.\underbrace{0 \ldots 0}_{k}\ \rho \ldots \rho > (\beta - 1)\overbrace{(\beta^{-1} + \beta^{-2} + \cdots + \beta^{-k})}^{1.0\ -\ 0\ldots01}$$

we have k zeros, p ρ and $\rho = \beta - 1$, from (1.2.1), we have the relative error

$$\begin{aligned}&\leq \frac{|\bar{y} - y|}{(\beta - 1)(\beta^{-1} + \beta^{-2} + \cdots + \beta^{-k})} \\ &< \frac{(\beta - 1)\beta^{-p}(\beta^{-1} + \beta^{-2} + \cdots + \beta^{-k})}{(\beta - 1)(\beta^{-1} + \beta^{-2} + \cdots + \beta^{-k})} = \beta^{-p} = 2\epsilon\end{aligned}$$

3° Case 3: $x - y < 1$ but $x - \bar{y} > 1$
If $x - \bar{y} = 1.\underbrace{0 \ldots 1}_{p} \Rightarrow x - y \geq 1$: a contradiction.

Because

$$|y - \bar{y}| < \beta^{-p} = 0.\underbrace{0 \ldots 1}_{p}$$

then

$$x - y = \underbrace{(1 + \beta^{-p})}_{x - \bar{y}} - \underbrace{|y - \bar{y}|}_{< \beta^{-p}} > 1$$

Proof is completed. ■

Conclusion: we can add "some" (not only 1) guard digits to make the relative error small enough. Especially when the two numbers are close to each other. To add one bit on the adder is very cheap.

Lecture 2

2026.03.01

1.3 Cancelation

Cancelatin is an important source of numerical instability. It occurs when we subtract two nearly equal numbers.

1.3.1 Catastrophic Cancelation

Example. b = 3.34, a = 1.22, c = 2.28, calculat $b^2 - 4ac$

- True value: $b^2 - 4ac = 0.0292$
- Calculated value: $b^2 \approx 11.2,\ 4ac \approx 11.1$, so $b^2 - 4ac \approx 0.1$

The Error $= 0.1 - 0.0292 = 0.0708$

- Computed answer $= 0.1 = 1.00 \times 10^{-1}$
- ulps $= 0.01 \times 10^{-1} = 10^{-3}$
- $0.0708 \approx 70$ ulps

The error is huge when we subtract two nearly equal numbers. This is called **catastrophic cancelation**. We can use **benign cancelation** to avoid this problem. For example, we use guard digits to get the relative error being small.

1.3.2 Benign Cancelation

However, guard digits can only reduce the error of operations between two exact number. In this case, b^2 and $4ac$ already contain errors. We can use **reformulation** to avoid this problem. For example,

Example.

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1.3.1}$$

If $b^2 \gg 4ac$, there is no problem for $\sqrt{b^2 - 4ac} \approx |b|$, but $-b + \sqrt{b^2 - 4ac}$ will cause catastrophic cancelation if $b > 0$.
We can reformulate by multiplying $-b - \sqrt{b^2 - 4ac}$ if $b > 0$.

$$\frac{2c}{-b + \sqrt{b^2 - 4ac}} \tag{1.3.2}$$

Now we can use (1.3.1) if $b < 0$ and (1.3.2) if $b > 0$ to avoid catastrophic cancelation.

Example. Calculate $x^2 - y^2$.

Assume $x \approx y$, $(x - y)(x + y)$ is much more accurate than $x^2 - y^2$ because x^2 and y^2 maybe rounded, so $x^2 - y^2$ may cause catastrophic cancelation. But, $x - y$ can be calculated by guard digits.

Remark. It is difficult to avoid all catastrophic cancelation but possible to remove some. A catastrophic cancellation is replaced by a benign cancellation.

Example. Calculate

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2} \tag{1.3.3}$$

a, b, c are the lengths of the triangle.

If $a \approx b + c$,

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

and $s - a$ may cause catastrophic cancelation. A new formula is by Kahan[1986]: for $a \geq b \geq c$,

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))} \tag{1.3.4}$$

Remark. Reformulation is a powerful tool to avoid catastrophic cancelation. It only works if guard digit is used.

1.3.3 Exactly Rounded Operations

Usually we choose to

$$\boxed{\text{Round}} \quad \Rightarrow \quad \boxed{\text{Compute}}$$

However, it may be not accurate. So, we introduce **exactly rounded** operations, which is to compute exactly then rounded to the nearest.

Definition (definition of rounding). There are two types of rounding:

Definition 1.3.1 (Rounding up). When the result's last digit $\geq$ half of the base, we round up, otherwise, we round down.

Definition 1.3.2 (Rounding even). Round to the closest value with even least significant digit.

Example. Round up

- $11.5 \Rightarrow 12$
- $12.5 \Rightarrow 13$

Round even

- $11.5 \Rightarrow 12$
- $12.5 \Rightarrow 12$

Theorem 1.3.1 (Reiser and Knuth). Let

$$x_0 = x, \quad x_1 = (x_0 \ominus y) \oplus y, \quad \cdots, \quad x_n = (x_{n-1} \ominus y) \oplus y$$

If $\oplus, \ominus$ is an exactly rounded using even rounding, then

$$x_n = x \ \forall n \text{ or } x_n = x_1 \ \forall n \geq 1$$

Example. $\beta = 10,\ p = 3,\ x = 1.00,\ y = -0.555$

$x - y = 1.555,\ x \ominus y = 1.56,\ (x \ominus y) + y = 1.56 - 0.555 = 1.005$

- Rounding up:

$$x_1 = (x \ominus y) \oplus y = 1.01, \quad x_1 - y = 1.565 \Rightarrow 1.57, \quad (x_1 \ominus y) + y = 1.57 - 0.555 = 1.015$$

$$x_2 = (x_1 \ominus y) \oplus y = 1.02$$

  Increase until $x_n = 9.45$.

- Rounding even:

$$x_1 = (x \ominus y) \oplus y = 1.00, \quad x_1 - y = 1.555 \Rightarrow 1.56, \quad (x_1 \ominus y) + y = 1.56 - 0.555 = 1.005$$

$$x_2 = (x_1 \ominus y) \oplus y = 1.00$$

Note. For the implementation, an array of words or floating-points used too much memory. Goldberg [1990] proposed a method to use only **two guard digits and one sticky bit**, the result is exact same as exactly rounded operations.

1.4 IEEE 754 Standard

1.4.1 Basics

IEEE 754 is the most used standard for floating-point arithmetic in computers nowadays. There are two of floating-point standard

- 754: specify $\beta = 2, p = 24$ for single precision, $\beta = 2, p = 53$ for double precision.
- 854: $\beta = 2$ or 10

Note. In standard 854, it accept $2, 10$ because

- $\beta = 2$ smaller β causes smaller relative error.
- $\beta = 10$ is most used.

Consider

$$\beta = 16, p = 1 \text{ versus } \beta = 2, p = 4$$

which both used 4 bits to store the significand, but there ϵ is different

$$\epsilon_{16} = \frac{16}{2} \cdot 16^{-1} = \frac{1}{2}, \quad \epsilon_2 = \frac{2}{2} \cdot 2^{-4} = \frac{1}{16}$$

However, $\beta = 16$ is used by IBM/370 due to two reasons:

1° Assume 4 bytes = 32 bits, $\beta = 16, p = 6$
significand:

$$4 \times 6 = 24 \text{ bits}$$

exponent:

$$32 - 24 - 1 = 7 \text{ bits (1 bit for sign)}$$

The range of exponent is $16^{-2^6} \to 16^{2^6} = 2^{2^8}$.
If we instead use $\beta = 2$ and to keep the same range of exponent, then

$$9 \text{ bits } (-2^8 \to 2^8) \text{ for exponent}$$

and

$$32 - 9 - 1 = 22 \text{ bits for significand}$$

Same exponents, less significand for $\beta = 2$ (24 vs. 22)

2° The cost of shifting: If $\beta = 16$, less frequently to adjust exponents when adding or subtracting two numbers. But nowadays the cost of shifting is not important since the computation speed.

1.4.2 IEEE standard: Significand and Exponent

For single precision, $\beta = 2, p = 24$ (23 bits as normalized), exponent 8 bits, and 1 bit for sign. Thus,

$$32 = 1 \text{ (sign bit) } + 8 \text{ (exponent) } + 23 \text{ (normalized significand)}$$

Example. $176.625 = 1.0101100101 \times 2^7$

$$0 \qquad 10000110 \qquad 01011001010000000000000$$

1 of $1.\cdots$ is not stored since the normalized.

Note. Here we use the bias of exponent

$$10000110 = 128 + 4 + 2 = 134,\ 134 - 127 = 7$$

Note that exponent may be negative, but here we don't use a sign bit for exponents

In this standard, it use rounding even

Binary	rounded	reason
10.00011	10.00	($< 1/2$, down)
10.00110	10.01	($> 1/2$, up)
10.11100	11.00	($1/2$, up)
10.10100	10.10	($1/2$, down)

Here is the summary of all standard:

IEEE	Fortran	C	Bits	Exp.	Mantissa
Single	REAL*4	float	32	8	24
Single-extended			44	≤ 11	32
Double	REAL*8	double	64	11	53
Double-extended	REAL*10	long double	$\geq$80	≥ 15	$\geq$64

Observe the single extended

$$44 \neq 11 \text{ (bits for exponent) } + 32 \text{ (bits for significand)}$$

Hardware implementation of extended precision normal don't use a hidden bit.
Minimal normalized positive number

$$1 \times 2^{-126} \approx 1.17 \times 10^{-38}$$

$e_{\min} = -126$

8 bits for exponent: 0 to 255. IEEE uses a biased approach for exponents

$$(0 \text{ to } 255) - 127 = -127 \text{ to } 128$$

There are some cases for exponent:

- -127: denormalized numbers, and 0
- -126 to 127: exponent for normalized numbers
- 128: special quantity (NaN, ∞)

Thus,

$$e_{\min} = -126, \quad e_{\max} = 127$$

> **Note.** The reason for not using $e_{\min} = -127$ is that $1/2^{e_{\min}}$ not overflow, $1/2^{e_{\max}}$ underflow.

1.4.3 IEEE standard: Extended Precision

Some may use more digits for intermediate calculations, which is called **extended precision**. For example, binary-decimal conversion, or some calculators use 13 digits for intermediate calculations but round to 10 digits for display.
Another example is that writing 9 digits is enough for short. Though $10^8 > 2^{24}$, 8 digits are not enough. From Section 2.1.2 of Goldberg [1990], in reading the 9-digit number, if extended precision is available, an efficient algorithm can be done so that the original binary representation is recovered.

1.4.4 IEEE standard: Exactly Rounded Operations

IEEE standard requires results of addition, subtraction, multiplication and division exactly rounded. The reason of doing so is to leave the portability of software.

Also, square root, remainder, conversion between integer and floating-point, internal formats and decimal are exactly rounded.

IEEE does not require transcendental functions to be exactly rounded (e.g. $\sin, \cos, \exp$). They cannot specify the precision because they are arbitrarily long

1.4.5 IEEE standard: Special Quantities

On some computer, (e.g. IBM 370) there is no special quantity, the value like $\sqrt{-4} = 2$ and it print a error message. But IEEE standard has special quantities

`NaN` (Not a Number)

which means not every pattern of bits is a valid number. Here is the summary of all cases:

Exponent	significand	represents
$e = e_{\min} - 1$	$f = 0$	$+0, -0$
$e = e_{\min} - 1$	$f \neq 0$	$0.f \times 2^{e_{\min}}$
$e_{\min} \leq e \leq e_{\max}$		$1.f \times 2^e$
$e = e_{\max} + 1$	$f = 0$	$\pm\infty$
$e = e_{\max} + 1$	$f \neq 0$	NaN

Sometimes even 0/0 occurs, the program can continue, which is better than stop and print an error message.

Example. Finding $f(x) = 0$, try different x's, even $0/0$ happens, other values may be ok.

If $b^2 - 4ac < 0$,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

return NaN, and $-b \pm \sqrt{b^2 - 4ac}$ return NaN as well.

Remark. In general when a NaN is in an operation, result is NaN

Here are some examples of producing NaN:

Operation	NaN by
$+$	$\infty + (-\infty)$
$\times$	$0 \times \infty$
$/$	$0/0, \infty/\infty$
REM	x REM 0, ∞ REM y
$\sqrt{}$	$\sqrt{x}$ when $x < 0$

1.4.6 IEEE standard: Infinity

$\beta = 10,\ p = 3,\ e_{\max} = 98,\ x = 3 \times 10^{70}$, x^2 overflow and replaced by ∞. Note that

$$0/0 = \text{NaN},\ 1/0 = \infty,\ -1/0 = -\infty \ \Rightarrow \text{ nonzero divided by 0 is } \infty \text{ or } -\infty$$

In the computaion, we replace ∞ with x, let $x \to \infty$.

Example.

$$\frac{x}{x^2+1} \text{ vs } \frac{1}{x + x^{-1}}$$

For $\dfrac{x}{x^2+1}$, if x is large, x^2 overflows, so $\dfrac{x}{\infty} = 0$ but not $\underbrace{\boxed{\dfrac{1}{x}}}_{\text{not } 0}$. For $\dfrac{1}{x + x^{-1}}$, if x is large, $\frac{1}{x}$ is ok, which is better.

When for $x = 0$,

$$\frac{1}{x + x^{-1}} = \frac{1}{0 + \infty} = \frac{1}{\infty} = 0$$

is also better.

Note. If no infinity arithmetic, an extra instruction is needed to test if $x = 0$. This may interrupt the pipeline.

1.4.7 IEEE standard: Signed Zero

To discuss why we need signed zero, first, it is available with 1 sign bit. If no signed zero, $1/(1/x)$ will fails when $x = \pm\infty$

$$x = \infty,\ 1/x = 0,\ 1/0 = +\infty$$
$$x = -\infty,\ 1/x = 0,\ 1/0 = +\infty$$

Definition 1.4.1 (Comparison of signed zero). In IEEE standard, $+0 = -0$.

± 0 is useful for some case

Example. We can define

$$\log x \equiv \begin{cases} -\infty & x = 0 \\ \texttt{NaN} & x < 0 \end{cases}$$

Definition 1.4.2 (Underflow). Underflow means a value smaller than the smallest floating-point number occurs.

When face a small underflow negative number, $\log x$ should return NaN. However, in the definition above,

$$\boxed{x \text{ underflow negative}} \quad \Rightarrow \quad \boxed{x \overset{\text{round}}{=} 0} \quad \Rightarrow \quad \boxed{\log x = -\infty}$$

With signed zero, we can define $\log x$ as

$$\log x \equiv \begin{cases} -\infty & x = +0 \\ \texttt{NaN} & x = -0 \\ \texttt{NaN} & x < 0 \end{cases}$$

Positive underflow will round to $+0$, and negative underflow will round to -0.

Example (Complex arithmetic).

$$\frac{1}{\sqrt{z}} \quad \text{and} \quad \sqrt{\frac{1}{z}}$$

If $z = -1$,

$$\frac{1}{\sqrt{-1}} = \frac{1}{i} = -i \quad \sqrt{\frac{1}{-1}} = i$$

Thus, $1/\sqrt{z} \neq \sqrt{1/z}$. Which happen because

$$i^2 = (-i)^2 = -1$$

However, by some restrictions (or ways of calculation), they can be equal.
If $z = -1 = -1 + 0i$,

$$\frac{1}{-1+0i} = \frac{-1-0i}{(-1)^2 + (0)^2} = -1 - 0i$$

then

$$\sqrt{\frac{1}{z}} = \sqrt{-1-0i} = -i$$

with signed zero, we can let $\sqrt{1/z} = 1/\sqrt{z}$.
But there is a disadvantage of signed zero, if $x = +0$ and $y = -0$, then

$$\frac{1}{x} = \infty, \quad \frac{1}{y} = -\infty$$

1.4.8 IEEE standard: Denormalized Numbers

Example. Consider

$$\beta = 10,\ p = 3,\ e_{\min} = -98,\ x = 6.87 \times 10^{-97},\ y = 6.81 \times 10^{-97}$$

x, y are ok but $x - y$ underflow and round to 0 even though $x \neq y$.

Note. It is important to preserve the property of

$$x = y \quad \Leftrightarrow \quad x - y = 0$$

with this case: `if (x != y) { z = 1/(x-y); }` The statement is true, but z becomes ∞.

IEEE use denormalized numbers to solve this problem, which is the most controversial part of IEEE standard. If denormalized number is used, 0.6×10^{-98} is also a floating-point number.

We do not store the leading 1 for $1.\cdots$. Recall the vaild range of exponent, $e \geq e_{\min}$, and we have $1.d\ldots d \times 2^e$. For denormalized numbers, we allow $e = e_{\min} - 1$, and the corresponding value is

$$0.d\ldots d \times 2^{e_{\min}}$$

Remark. The reason why not using

$$1.d\ldots d \times 2^{e_{\min}-1}$$

is that then we cannot represent

$$0.0d\ldots d \times 2^{e_{\min}}$$

which is even smaller than the smallest normalized number.

Example. $6.87 \times 10^{-97} - 6.81 \times 10^{-97} = 0.06 \times 10^{-97}$

is underflow due to the cancelation.

Here is an example of using denormalized numbers:

Example.

$$\frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{(c+di)(c-di)} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i$$

If c or $d > \sqrt{\beta}\beta^{e_{\max}/2}$, there will be overflow.

Definition 1.4.3 (Overflow). Overflow means a value larger than the largest floating-point number occurs.

Lemma 1.4.1 (Smith's lemma).

$$\frac{a+bi}{c+di} = \begin{cases} \frac{a+b(d/c)}{c+d(d/c)} + \frac{b-a(d/c)}{c+d(d/c)}i & \text{if } (|d| < |c|) \\ \frac{b+a(c/d)}{d+c(c/d)} + \frac{-a+b(c/d)}{d+c(c/d)}i & \text{if } (|d| \geq |c|) \end{cases}$$

This is a reformulation to avoid overflow. However, using Smith's formula, some issues may occur without denormalized numbers:
If

$$a = 2 \times 10^{-98},\ b = 1 \times 10^{-98},\ c = 4 \times 10^{-98},\ d = 2 \times 10^{-98}$$

then

$$\frac{d}{c} = 0.5, \quad c + d \cdot \frac{d}{c} = 5 \times 10^{-98}$$

$$b \cdot \frac{d}{c} = 1 \times 10^{-98} \times 0.5 = \mathbf{0}, \quad a + b \cdot \frac{d}{c} = 2 \times 10^{-98}$$

Solution $= 0.4$, which is wrong.

If denormalized numbers are used, 0.5×10^{-98} can be stored

$$a + b \cdot \frac{d}{c} = 2 \times 10^{-98} + 0.5 \times 10^{-98} = 2.5 \times 10^{-98}$$

Solution $= 0.5$, which is correct.

Note. Usually hardware does not support denormalized numbers directly, we use software to simulate it.

Note. Program may be slow when there is a lot of underflow.

1.5 Trap Handlers

1.5.1 Exception, Flags, Trap handlers

When we face with some special situations (e.g. divide by zero), we have to know what happens. For this purpose, IEEE requires vendors to provide a way to get status flags. **overflow**, **underflow**, **division by zero**, **invalid operation**, **inexact**.

Definition (IEEE exception)**.** IEEE defines five exceptions:

Definition 1.5.1 (Overflow)**.** Larger than the maximal floating-point number.

Definition 1.5.2 (Underflow)**.** Smaller than the minimal floating-point number.

Definition 1.5.3 (Division by zero)**.** Nonzero divided by zero.

Definition 1.5.4 (Invalid operation)**.** $\infty + (-\infty)$, $0 \times \infty$, $0/0$,, ∞/∞, x REM 0, ∞ REM y, $\sqrt{x}, x < 0$, any comparison involves a `NaN`.

Remark. Invalid returns `NaN`; But `NaN` may not be from invalid operations.

Definition 1.5.5 (Inexact)**.** The result is not exact: $\beta = 10, p = 3, 3.5 \times 4.2 = 14.7$ exact, $3.5 \times 4.3 = 15.05 \Rightarrow 15.0$ not exact.

Remark. Inexact exception is raised so often, usually we ignore it.

Here is the summary of all exceptions and their corresponding handlers:

Exception	when trap disabled	argument to handler
overflow	$\pm\infty$ or $\pm 1.1\cdots 1 \times 2^{e_{\max}}$	round$(x \times 2^{-\alpha})$
underflow	$0, \pm 2^{e_{\min}}$, or denormalized	round$(x \times 2^{\alpha})$
division by zero	∞	operands
invalid	NaN	operands
inexact	round(x)	round(x)

Definition 1.5.6 (Trap handler). Special subroutines to handle exception, which can be designed by users.

In the above table, " when trap disabled"means results of operations if trap handlers not used.

Example. $\alpha = 192$, for single precision, $\alpha = 1536$ for double precision. We can't store x since it is too large/small.

1.5.2 Compiler Operations

Compiler may provide a way so the program stops if an exception occurs. For example, SUN's C compiler provides a command

```
-ftrap=t
```

t: `%all, %none, common, [no%]invalid, [no%]overflow, [no%]underflow, [no%]division, [no%]inexact`

Note. common: invalid, division by zero, and overflow

when compile you will see

```
Note: IEEE floating-point exception flags raised:
    Inexact;  Underflow;
See the Numerical Computation Guide, ieee_flags(3M)
```

Note. gcc doesn't have the functionality to explicitly detect exceptions. It only provides

- `-fno-trapping-math`: The default if `-ftrapping-math`. Setting this option may allow faster code if one relies on " non-stop"IEEE arithmetic.
- `-ftrapv`: Generates traps for signed overflow on addition, subtraction, multiplication.

1.5.3 Trap Handlers

For example,

```
do {
...
} while { not x >= 100; }
```

If `x = NaN`, an infinite loop will occur. Any comparison involving `NaN` is wrong. Thus `x >= 100` returns `false`. A trap handler can be installed to abort it.

Another example is that calculate $x_1 \times \cdots \times x_n$ may overflow in the middle, but fine in the end.

```
for (i = 1; i <= n; i++)
    p = p * x[i];
```

If trap handlers are not used, the result will be ∞ and wrong, i.e.

$$x_1 \times \cdots \times x_r, r \leq n \text{ overflow }, \quad x_1 \times \cdots \times x_n \text{ in the range}$$

> **Note.** There might be a solution,
>
> $$e^{\Sigma \log(x_i)}$$
>
> but quite unaccurate and cost more.

A trap handler can be installed to save the intermediate result

```
for (i = 1; i <= n; i++) {
    if (p * x[i] overflow) { // trap handler to catch overflow
        p = p * pow(10, -a);
        count = count + 1 ;
    }
    p = p * x[i];
}
p = p * pow(10, a*count) ;
```

Lecture 3

On SUN machines, there is an additional math library: `libsunmath.a`. In this library, there are `exp2`, `exp10`, ..., `ieee_flags`, `ieee_handler`, and `ieee_retrospective`. 2026.03.07

```
#include <stdio.h>
#include <sys/ieeefp.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned    code, addr;

    code = sip->si_code;
    addr = (unsigned) sip->si_addr;
    fprintf(stderr, "fp exception %x at address %x \n", code, addr);
}
int main()
{
    double  x;

    /* trap on common floating point exceptions */
    if (ieee_handler("set", "common", handler) != 0)
```

```
        printf("Did not set exception handler \n");

    /* cause an underflow exception (not reported) */
    x = min_normal();
    printf("min_normal = %g \n", x);
    x = x / 13.0;
    printf("min_normal / 13.0 = %g \n", x);

    /* cause an overflow exception (reported) */
    x = max_normal();
    printf("max_normal = %g \n", x);
    x = x * x;
    printf("max_normal * max_normal = %g \n", x);

    ieee_retrospective(stderr);
    return 0;
}
```

The results of the above code are as follows:

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
 Note: IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),  ieee_handler(3M)
```

In this example, we can see that the underflow exception is not reported, due to the denormalized numbers in the IEEE 754 standard. In this handler, we simply print the something, but in practice, we can do more error handling in this kind of subroutine.

Example. Calculate x^n, n: integer.

```
double pow(double x, int n)
{
    double tmp = x, ret = 1.0;
    for(int t=n; t>0; t/=2)
    {
        if (t % 2 == 1) ret *= tmp;
        tmp = tmp * tmp;
    }
    return ret;
}
```

When $n > 0$, the above code can do

$$x^n = \prod_{i=0}^{\lfloor \log_2 n \rfloor} x^{2^i \cdot (n \mod 2^{i+1} > 0)}$$

to calculate x^n in $O(\log n)$ time. But if $n < 0$, we need to use

$$x^n = \left(\frac{1}{x}\right)^{-n} = \frac{1}{x^{-n}}$$

Due to there is already error on $1/x$, `pow(1/x, -n)` is less accurate than `1/pow(x, -n)`.
However, there is another problem: if `pow(x, -n)` cause an underflow, but `1/pow(x, -n)` does not overflow

$$e_{\min} = -126, \quad 2^{-e_{\min}} = 2^{126} < 2^{127} = 2^{e_{\max}}$$

The solution is

1° turn off overflow & underflow trap enable bits, save overflow & underflow status bits.

2° Calculate `pow(x, -n)`.

- If neither overflow nor underflow status is set ⇒ restore them.
- If one is set, restore & calculate `pow(1/x, -n)`, which causes correct exception to occur restore them.

But practically, the calculation of `pow()` is more complicated.

Example. Calculate arccos using arctan.

Theorem 1.5.1.

$$\arccos x = 2\arctan\sqrt{\frac{1-x}{1+x}}$$

Proof.

$$\cos\theta = x = 2\cos^2\frac{\theta}{2} - 1 = 1 - 2\sin^2\frac{\theta}{2}$$

$$\Rightarrow \quad \cos\frac{\theta}{2} = \sqrt{\frac{1+x}{2}}, \quad \sin\frac{\theta}{2} = \sqrt{\frac{1-x}{2}}$$

$$\Rightarrow \quad \tan\frac{\theta}{2} = \sqrt{\frac{1-x}{1+x}} \quad \Rightarrow \quad \theta = 2\arctan\sqrt{\frac{1-x}{1+x}}$$

■

Consider $x = -1$

$$\arctan(\infty) = \frac{\pi}{2} \quad \Rightarrow \quad \arccos(-1) = \pi$$

There is a small problem $(1-x)/(1+x)$ cause the divide-by-zero flag set though $\arccos(-1)$ is not exceptional. The solution is to save divide-by-zero flag, restore it after arccos computation.

1.6 Real Examples

1.6.1 Same Code but Different Architectures

```c
#include <stdio.h>

int main()
{
    float a = 123.123;
    printf("%.10f\n", a);
    printf("%.10f\n", a*a);

    a = 123.125;
    printf("%.10f\n", a);
    printf("%.10f\n", a*a);
}
```

```
$ gcc test.c;./a.out
123.1230010986
15159.2734375000
123.1250000000
15159.7656250000
$ gcc -m32 test.c;./a.out
123.1230010986
15159.2733995339
123.1250000000
15159.7656250000
```

> **Note.** `-m 32` generates code for a 32-bit environment.

Same code gives different results under 32 and 64-bit environments.

> **Note.** On 32 bit, 387 floating-point coprocessor is used. From gcc manual, "The temporary results are computed in 80-bit precision instead of the precision specified by the type, resulting in slightly different results compared to most of other chips." In other words, they somehow violate IEEE standard.
>
> The number 123.123 has infinite digits after transformed to binary, assigning the compiler option can make the result consistent.

> **Example.** `-mfpmath=387` to let the 64-bit machine run like a 32-bit one.

```
$ gcc -mfpmath=387 test.c;./a.out
123.1230010986
15159.2733995339
123.1250000000
15159.7656250000
```

Example. `-ffloat-store` make the 32-bit machine like a 64-bit one Manual of the option, "Do not store floating-point variables in registers, and inhibit other options that might change whether a floating-point value is taken from a register or memory."

```
$ gcc -ffloat-store test.c;./a.out
123.1230010986
15159.2734375000
123.1250000000
15159.7656250000
$ gcc -ffloat-store -m32 test.c;./a.out
123.1230010986
15159.2734375000
123.1250000000
15159.7656250000
```

1.6.2 Order of Operations

For the same code, other issues such as order of operations can also affect results.

Example. Consider running a real example using a machine learning software LIBSVM.

In the O0 optimization level:

```
$ g++ -O0 svm-train.c svm.cpp -o svm-train -lm
$ ./svm-train -c 100 -e 0.00001 heart_scale
........*..*
optimization finished, #iter = 2872
nu = 0.148045
obj = -2526.925470, rho = 1.145512
nSV = 107, nBSV = 9
Total nSV = 107
```

In the Ofast optimization level:

```
$ g++ -Ofast svm-train.c svm.cpp -o svm-train -lm
$ ./svm-train -c 100 -e 0.00001 heart_scale
........*..*
optimization finished, #iter = 2910
nu = 0.148045
obj = -2526.925470, rho = 1.145510
nSV = 107, nBSV = 9
Total nSV = 107
```

Some compiler optimizations may change the order of operations to improve performance. On default settings for 64-bit environments, O0 to O3 produce the same results. But from `gcc` manual, `-Ofast` "disregards strict standards compliance".
`-mfpmath=387` is even more sensitive to optimizations. To produce the same results with `-mfpmath=387`, we need to disable all optimizations due to more complicated interactions with registers and memory.

Chapter 2

Direct Methods for Linear Systems

2.1 LU Factorization

2.1.1 Solving Dense Linear Systems

Solving a linear system $Ax = b$ is an important problem in numerical method. In this kind of problem, we assume A is a dense matrix, i.e. most

$$A_{ij} \neq 0$$

and we allocate space to store

$$A_{ij}, \quad \forall i, \ \forall j$$

2.1.2 Triangular Systems

Start from a triangular system,

$$\begin{pmatrix} l_{11} & \\ l_{21} & l_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

if $l_{11}, l_{22} \neq 0$,

$$x_1 = \frac{b_1}{l_{11}}, \qquad x_2 = \frac{b_2 - l_{21}x_1}{l_{22}}$$

Theorem 2.1.1. In general,

$$x_i = \frac{\left(b_i - \sum_{j=1}^{i-1} l_{ij}x_j \right)}{l_{ii}}$$

This is called "forward substitution", which requires $O(n^2)$ operations. Similarly, we can solve an upper triangular system using "backward substitution".

Theorem 2.1.2. In general,

$$x_i = \frac{\left(b_i - \sum_{j=i+1}^{n} u_{ij}x_j \right)}{u_{ii}}$$

2.1.3 LU Factorization

Example. Consider the following system

$$\begin{aligned} 3x_1 + 5x_2 &= 9 \\ 6x_1 + 7x_2 &= 4 \end{aligned}$$

By Gaussian elimination,

$$\begin{aligned} 3x_1 + 5x_2 &= 9 \\ -3x_2 &= -14 \end{aligned}$$

Actually, the process is in the below

$$\begin{pmatrix} 3 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} 3 & 5 \\ 0 & -3 \end{pmatrix}$$

$$\Rightarrow \quad \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 9 \\ 4 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 9 \\ -14 \end{pmatrix}$$

Then solve

$$\begin{pmatrix} 3 & 5 \\ 0 & -3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 9 \\ -14 \end{pmatrix}$$

is same as solving the original system.

Example. Consider

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{pmatrix}$$

We do the Gaussian Transformation to get the upper triangular matrix, if

$$M_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 0 & 1 \end{pmatrix}$$

then

$$M_1 A = \begin{pmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{pmatrix}$$

Note that

$$\begin{pmatrix} -3 & -6 \\ -6 & -11 \end{pmatrix} = \begin{pmatrix} 5 & 8 \\ 6 & 10 \end{pmatrix} - \begin{pmatrix} 2 \\ 3 \end{pmatrix} \begin{pmatrix} 4 & 7 \end{pmatrix}$$

Likewise,

$$M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix}$$

$$\Rightarrow \quad M_2(M_1 A) = \begin{pmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{pmatrix}$$

Hence, the whole Gaussian elimination process is

$$U = M_{n-1} \cdots M_2 M_1 A$$

$$A = (M_1^{-1}(M_2^{-1} \cdots (M_{n-1}^{-1}U))) = LU$$

M_i^{-1} is a lower triangular matrix,

$$M_i = \begin{pmatrix} 1 & 0 & 0 \\ \vdots & \ddots & 0 \\ m_{i1} & \cdots & 1 \end{pmatrix}, \quad M_i^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ \vdots & \ddots & 0 \\ -m_{i1} & \cdots & 1 \end{pmatrix}$$

$$M_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 0 & 1 \end{pmatrix} \quad \Rightarrow \quad M_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix}$$

$$M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \quad \Rightarrow \quad M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$$

$$M_1^{-1}M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix}$$

In the implementation, we do not store $M_1, \ldots, M_{n-1}$, they can be stored in the lower triangular part of A.

The reason of choosing LU factorization than Gaussian elimination is that Gaussian elimination overwrites the original matrix. If you do not overwrite A, need another array (the same size as A and $L + U$)

Gaussian elimination update right-hand side together. To handle multiple right-hand sides, LU is the most effective way. The algorithm is as follows,

```
for k=1:n-1
    A(k+1:n,k) = A(k+1:n,k)/A(k,k);
    A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - A(k+1:n,k)*A(k,k+1:n);
end
```

The number of operations is

$$\approx \sum_{k=1}^{n-1} 2k^2 \approx 2 \cdot \frac{n(n-1)(2n-1)}{6} = \frac{2}{3}n^3$$

> **Note.** The constant 2 is due to one multiplication and one subtraction.

2.1.4 Problem of LU Factorization

If matrix A has a zero diagonal such as

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

then we can't generate M_1. Even if we can do LU factorization, numerical error is sometimes large.

> **Example.** Assume $\beta = 10, p = 3$, no guard digit. Solving
>
> $$\begin{pmatrix} 0.001 & 1.00 \\ 1.00 & 2.00 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1.00 \\ 3.00 \end{pmatrix}$$

After LU factorization, we have

$$L = \begin{pmatrix} 1 & 0 \\ 1000 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 0.001 & 1 \\ 0 & -1000 \end{pmatrix}$$

$2 - 1000 = -998 \approx -1000$ in the floating-point arithmetic. However, the real solution is

$$\begin{pmatrix} 0.001 & 1 \\ 0 & -998 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -997 \end{pmatrix}$$

$$x_2 = \frac{997}{998}$$

$$x_1 = 1000(1 - \frac{997}{998}) = \frac{1000}{998}$$

But using our system, we have

$$\begin{pmatrix} 1 & 0 \\ 1000 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1.00 \\ 3.00 \end{pmatrix}$$

$$y_1 = 1.00, \quad y_2 = 3.00 - 1000 \approx -1000$$

$$\begin{pmatrix} 0.001 & 1 \\ 0 & -1000 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1.00 \\ -1000 \end{pmatrix} \tag{2.1.1}$$

Then we get

$$x_1 = 0, \quad x_2 = 1$$

$(0, 1)$ is very different from the real solution $(1000/998, 997/998)$.

Remark. A small pivot may cause large numerical errors.

In the eqs. (2.1.1), if x_2 contains an errors, because $1 \gg 0.001$, x_1 is seriously affected. Hence, if we interchange the two rows of A

$$\begin{pmatrix} 0.001 & 1 \\ 1 & 2 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 1 & 2 \\ 0.001 & 1 \end{pmatrix}$$

then

$$L = \begin{pmatrix} 1 & 0 \\ 0.001 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 0.001 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 3.00 \\ 1.00 \end{pmatrix}$$

$$y_1 = 3.00, \quad y_2 = 1.00$$

therefore, we solve

$$\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3.00 \\ 1.00 \end{pmatrix}$$

$$x_1 = 1.00, \quad x_2 = 1.00$$

which is much closer to the real solution.

Remark. We can use permutation matrix P to represent the row interchanges.

Example.

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

This means we do the exchange of

$$4 \to 1, \quad 1 \to 2, \quad 3 \to 3, \quad 2 \to 4$$

2.1.5 Pivoting Strategies

Pivoting in LU:

Example.

$$A = \begin{pmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{pmatrix}$$

We first find the largest absolute value in the first column: 6, so we do

$$P_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

then

$$P_1 A = \begin{pmatrix} 6 & 18 & -12 \\ 2 & 4 & -2 \\ 3 & 17 & 10 \end{pmatrix} \quad \Rightarrow \quad M_1 = \begin{pmatrix} 1 & 0 & 0 \\ -1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix}$$

$$M_1 P_1 A = \begin{pmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 4 \end{pmatrix}$$

$|8| > |-2|$

$$P_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

then

$$M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/4 & 1 \end{pmatrix}$$

$$M_2 P_2 M_1 P_1 A = \begin{pmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{pmatrix}$$

The general form is

$$M_{n-1} P_{n-1} \cdots M_2 P_2 M_1 P_1 A = U$$

Now we have

$$Ax = b$$

$$\Rightarrow \quad (M_{n-1} P_{n-1} \cdots M_2 P_2 M_1 P_1)^{-1} Ax = (M_{n-1} P_{n-1} \cdots M_2 P_2 M_1 P_1)^{-1} b$$

hence, we can solve

$$Ux = (M_{n-1} P_{n-1} \cdots M_2 P_2 M_1 P_1)^{-1} b$$

2.1.6 Pivoting Strategies: Implementation

However storing M_i and P_i is not efficient, we can try to overwrite A to store M_i and P_i.

$$\begin{pmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{pmatrix} \xrightarrow{p(1)=3} \begin{pmatrix} 6 & 18 & -12 \\ 2 & 4 & -2 \\ 3 & 17 & 10 \end{pmatrix}$$

then

$$\begin{pmatrix} 6 & 18 & -12 \\ \boxed{1/3} & -2 & 2 \\ \boxed{1/2} & 8 & 16 \end{pmatrix} \xrightarrow{p(2)=3} \begin{pmatrix} 6 & 18 & -12 \\ 1/2 & 8 & 16 \\ 1/3 & -2 & 2 \end{pmatrix}$$

Remark. For the boxed elements, we usually fill 0 in the upper triangular part, but to store the multiplier if that row more efficient. We can do it by filling the multiplier in the lower triangular part (dividing it by the pivot in its column).

In the above procedure we use an array p to store the permutation. Then we try to swap two rows of A together.

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1/3 & -1/4 & 1 \end{pmatrix} \begin{pmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{pmatrix}$$

Hence, it is possible to get the general form

$$PA = LU$$

where P is the permutation matrix. Now we can overwrite A to store L and U, and use an array p to store the permutation. To get x, we habe

$$PAx = Pb$$

then we solve

$$(LU)x = Pb$$

We can get the general form of the algorithm as follows,

```
for k=1:n-1
    find m
    swap A(k,1:n) and A(m, 1:n)
    p(k) = m;
    A(k+1:n,k) = A(k+1:n,k)/A(k,k);
    A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - A(k+1:n,k)*A(k,k+1:n);
end
```

Theorem 2.1.3.

$$PA = LU$$

and L is the lower-triangular matrix obtained in the algorithm.

Proof. By the definition of P, we hope to get

$$\begin{aligned} PA &= P_{n-1} \cdots P_2 P_1 A \\ &= P_{n-1} \cdots P_1^{-1} M_1^{-1} \cdots P_{n-1}^{-1} M_{n-1}^{-1} U \\ &= LU \end{aligned}$$

Consider i-th column of

$$P_{n-1}\cdots P_1^{-1}M_1^{-1}\cdots P_{n-1}^{-1}M_{n-1}^{-1}$$

Note that for

$$P_j^{-1},\ M_j^{-1},\quad j>i$$

the i-th column is always

$$\begin{pmatrix} & \vdots & \\ & 0 & \\ \cdots & 1 & \cdots \\ & 0 & \\ & \vdots & \end{pmatrix}$$

Thus,

$$(P_{n-1}\cdots P_1P_1^{-1}M_1^{-1}\cdots P_{n-1}^{-1}M_{n-1}^{-1})_{:,i}=(P_{n-1}\cdots P_1^{-1}M_1^{-1}\cdots P_i^{-1}M_i^{-1})_{:,i}$$

Note.

$$(M_i^{-1})_{:,i}=\begin{pmatrix}0\\ \vdots\\ 0\\ 1\\ m_{i+1,i}\\ \vdots\\ m_{n,i}\end{pmatrix}$$

So, by similar argument, we have

$$(P_{n-1}\cdots P_1P_1^{-1}M_1^{-1}\cdots P_i^{-1}M_i^{-1})_{:,i}=(P_{n-1}\cdots P_{i+1}M_i^{-1})_{:,i} \tag{2.1.2}$$

Here is the detail

$$(P_i^{-1}M_i^{-1})=\begin{pmatrix}\vdots\\ 0\\ x'\\ x'\\ \vdots\\ x'\end{pmatrix}$$

because P_i only swap two rows in $i,\ldots,n$-th rows, then

$$(M_{i-1}^{-1}P_i^{-1}M_i^{-1})_{:,i}=\begin{pmatrix} & 1 & & & \\ & x & 1 & & \\ \ldots & \vdots & 0 & 1 & \\ & \vdots & \vdots & & \ddots \\ & x & 0 & & \end{pmatrix}\begin{pmatrix}\vdots\\ 0\\ x'\\ x'\\ \vdots\\ x'\end{pmatrix}=\begin{pmatrix}\vdots\\ 0\\ x'\\ x'\\ \vdots\\ x'\end{pmatrix}$$

which does not change the i-th column.

Therefore, we have

$$(P_{n-1}\cdots P_1P_1^{-1}M_1^{-1}\cdots P_i^{-1}M_i^{-1})_{:,i} = (P_{n-1}\cdots P_1P_1^{-1}P_2^{-1}\cdots P_i^{-1}M_i^{-1})_{:,i}$$
$$= (P_{n-1}\cdots P_{i+1}M_i^{-1})_{:,i}$$

where eliminate $P_i^{-1} = P_i$. ■

Note. Eqs. (2.1.2) is exactly get in the algorithm, and the $P_{n-1}, \ldots, P_{i+1}$ swaping them into the corresponding position.

2.2 Cholesky Factorization

There are some special cases of matrix A. For example, diagonal, tri-diagonal, banded matrices, positive definite matrices, etc.

Example. A banded matrix,

$$\begin{pmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 7 & 0 & 0 \\ 8 & 9 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 7 & 8 \\ 0 & 0 & 9 & 1 & 2 & 3 \\ 0 & 0 & 0 & 4 & 5 & 6 \end{pmatrix}$$

Many numerical methods are designed for special matrices such as symmetric positive definite matrices.

Definition 2.2.1 (positive definite). An $n \times n$ matrix A is positive definite if

$$x^TAx > 0, \quad \forall x \in \mathbb{R}^n,\ x \neq 0$$

Theorem 2.2.1. A is symmetric positive definite if and only if all A's eigenvalues are positive.

Theorem 2.2.2 (symmetric positive definite). A is SPD $\Rightarrow$ all diagonal elements are positive, all principle sub-matrices are positive definite.

Definition 2.2.2 (principle sub-matrix). Let A be an $n \times n$ matrix. A principal submatrix is a submatrix obtained by deleting some rows and the same columns. Formally, given a subset of indices $I \subseteq \{1, 2, \ldots, n\}$, the principal submatrix A_I is formed by selecting the rows and columns indexed by I.

Theorem 2.2.3 (Cholesky factorization). If A is SPD, then there exists a unique lower triangular matrix L such that

$$A = LL^T$$

The above theroem is called Cholesky factorization.

2.2.1 Cholesky Factorization

Theorem 2.2.4 (Cholesky factorization). We have

$$\begin{aligned} A &= \begin{pmatrix} \alpha & v^T \\ v & B \end{pmatrix} \qquad (\alpha > 0) \\ &= \begin{pmatrix} \sqrt{\alpha} & 0 \\ \frac{v}{\sqrt{\alpha}} & I \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & \frac{v^T}{\sqrt{\alpha}} \\ 0 & B - \frac{vv^T}{\alpha} \end{pmatrix} \\ &= \begin{pmatrix} \sqrt{\alpha} & 0 \\ \frac{v}{\sqrt{\alpha}} & I \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & 0 \\ 0 & B - \frac{vv^T}{\alpha} \end{pmatrix} \begin{pmatrix} 1 & \frac{v^T}{\alpha} \\ 0 & I \end{pmatrix} \end{aligned}$$

If

$$B - \frac{vv^T}{\alpha} = \bar{L}\bar{L}^T$$

i.e. it is also SPD, then we have

$$\begin{aligned} \begin{pmatrix} \sqrt{\alpha} & 0 \\ \frac{v}{\sqrt{\alpha}} & I \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & 0 \\ 0 & \bar{L}\bar{L}^T \end{pmatrix} \begin{pmatrix} 1 & \frac{v^T}{\alpha} \\ 0 & I \end{pmatrix} &= \left(\begin{pmatrix} \sqrt{\alpha} & 0 \\ \frac{v}{\sqrt{\alpha}} & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \bar{L} \end{pmatrix}\right)\left(\begin{pmatrix} 1 & 0 \\ 0 & \bar{L}^T \end{pmatrix} \begin{pmatrix} 1 & \frac{v^T}{\alpha} \\ 0 & I \end{pmatrix}\right) \\ &= \begin{pmatrix} \sqrt{\alpha} & 0 \\ \frac{v}{\sqrt{\alpha}} & \bar{L} \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & \frac{v^T}{\sqrt{\alpha}} \\ 0 & \bar{L}^T \end{pmatrix} \end{aligned}$$

Note. If we can proof that $B - \frac{vv^T}{\alpha}$ is also SPD, then the same procedure can be applied to $B - \frac{vv^T}{\alpha}$, and we can get the Cholesky factorization of A.

Lemma 2.2.1. If A is SPD, then $B - \frac{vv^T}{\alpha}$ is also SPD.

Proof. For the symmertic, it is obvious. For the positive definiteness, we have for any $x \neq 0$,

$$\begin{aligned} x^T(B - \frac{vv^T}{\alpha})x &= x^TBx - \frac{(v^Tx)^2}{\alpha} \\ &= \begin{pmatrix} -\frac{v^Tx}{\sqrt{\alpha}} & x^T \end{pmatrix} \begin{pmatrix} 0 \\ -\frac{v^Tx}{\alpha}v + Bx \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} -\frac{v^Tx}{\sqrt{\alpha}} & x^T \end{pmatrix}}_{\mathbf{v}} \underbrace{\begin{pmatrix} \alpha & v^T \\ v & B \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} -\frac{v^Tx}{\sqrt{\alpha}} \\ x \end{pmatrix}}_{\mathbf{v}^T} > 0 \qquad \text{(because } A \text{ is SPD)} \end{aligned}$$

■

Now, we can go through the implementation in the **outer product form**.

```
for k=1:n
    A(k,k) = sqrt(A(k,k))
    A(k+1:n,k) = A(k+1:n,k)/A(k,k)
    for j=k+1:n
        A(j:n,j) = A(j:n,j) - A(j:n,k)A(j,k)
    end
end
```

The inner loop comes from

$$B_{:,j} - (v/\alpha)_j \cdot \frac{v}{\sqrt{\alpha}}$$

There is another implementation way, treat

$$A = LL^T$$

as a system of equations with unknowns L and solve it.

$$A_{ij} = \sum_{k=1}^{n} L_{ik}(L^T)_{kj} = \sum_{k=1}^{n} L_{ik}L_{jk} = \sum_{k=1}^{n} L_{jk}L_{ik}$$

$$\Rightarrow \quad A_{:,j} = \sum_{k=1}^{n} L_{jk}L_{:,k}$$

Because L is lower triangular, we have

$$L_{j,j+1} = \cdots = L_{j,n} = 0$$

Thus,

$$A_{:,j} = \sum_{k=1}^{j} L_{jk}L_{:,k}$$

$$L_{jj}L_{:,j} = A_{:,j} - \sum_{k=1}^{j-1} L_{jk}L_{:,k}$$

The $(j-1)$-th column is already computed, so we can compute $L_{:,j}$ by let

$$L_{jj}L_{jj} = A_{j,j} - \sum_{k=1}^{j-1} L_{jk}L_{j,k}$$

Then

$$L_{j+1:n,j} = \frac{(A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{jk}L_{j+1:n,k})}{L_{jj}}$$

In practical implementation we calculate

$$A_{j:n,j} - \sum_{k=1}^{j-1} L_{jk}L_{j:n,k}$$

first.

```
for j=1:n
    v(j:n) = A(j:n,j)
    for k=1:j-1
        v(j:n) = v(j:n) - L(j,k)L(j:n,k)
    end
    L(j:n,j) = v(j:n)/sqrt(v(j))
end
```

> **Note.**
>
> ```
> v(j:n) = v(j:n) - L(j,k)L(j:n,k)
> ```
>
> is a **apxy** operation.

For the time complexity,

$$\begin{aligned}\sum_{j=1}^{n}\sum_{k=1}^{j-1}2(n-j+1) &= 2\sum_{j=1}^{n}(j-1)(n-j+1)\\ &= \sum_{j=1}^{n}(nj-n-j^2+j+j-1)\\ &\approx 2(n\cdot\frac{n(n+1)}{2}-\frac{n(n+1)(2n+1)}{6}) = O\left(\frac{n}{3}\right)\end{aligned}$$

Which is half of the time complexity of LU factorization.

Lecture 4

2.3 Condition of Matrices

2026.03.14

2.3.1 Vector and Matrix Norms

To calculate the error, we can separate into 3 types: scalar, vector, and matrix. For scalar, we can simply use

- absolute error:
$$|\tilde{\alpha}-\alpha|$$
- relative error:
$$\frac{|\tilde{\alpha}-\alpha|}{|\alpha|}$$

To evaluate the vector and matrix error, we need to define the vector and matrix norms, which is a kind of metric.

Definition (Vector Norm)**.** Here are some common vector norms:

Definition 2.3.1 (l-norm)**.**
$$\|x\|_l = \left(\sum_{i=1}^{n}|x_i|^l\right)^{\frac{1}{l}}$$

for instance,

Definition 2.3.2 (1-norm)**.**
$$\|x\|_1 = \sum_{i=1}^{n}|x_i|$$

Definition 2.3.3 (∞-norm)**.**
$$\|x\|_\infty \equiv \lim_{l\to\infty}\|x\|_l$$

Theorem 2.3.1.
$$\|x\|_\infty = \max_{1\le i\le n}|x_i|$$

Proof. We can induce by Squeeze Theorem. For any l, we have

$$\sqrt[l]{(\max|x_i|)^l} \le \sqrt[l]{|x_1|^l + \cdots + |x_n|^l} \le \sqrt[l]{n(\max|x_i|)^l}$$

which implies

$$\max|x_i| \le \|x\|_l \le n^{\frac{1}{l}} \max|x_i|$$

By Squeeze Theorem ($l \to \infty$), we have

$$\|x\|_\infty = \max|x_i|$$

Proof is complete. ■

Example.

$$x = \begin{pmatrix} 1 \\ 1000 \\ 9 \end{pmatrix}, \quad x = \begin{pmatrix} 1.1 \\ 99 \\ 11 \end{pmatrix}$$

$$\|\hat{x} - x\|_\infty = 2, \quad \frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty} = 0.02, \quad \frac{\|\hat{x} - x\|_\infty}{\|\hat{x}\|_\infty} = 0.0202$$

$$\|\hat{x} - x\|_2 = 2.238, \quad \frac{\|\hat{x} - x\|_2}{\|x\|_2} = 0.0223, \quad \frac{\|\hat{x} - x\|_2}{\|\hat{x}\|_2} = 0.0225$$

Corollary 2.3.1. For l-norm, all norms are equivalent to each other, which means there exist $c_1, c_2 > 0$ such that

$$c_1\|x\|_{l_1} \le \|x\|_{l_2} \le c_2\|x\|_{l_1}$$

for all $x \in \mathbb{R}^n$.

For instance,

$$\|x\|_2 \le \|x\|_1 \le \sqrt{n}\|x\|_2 \tag{2.3.1}$$

$$\|x\|_\infty \le \|x\|_2 \le \sqrt{n}\|x\|_\infty \tag{2.3.2}$$

$$\|x\|_\infty \le \|x\|_1 \le n\|x\|_\infty \tag{2.3.3}$$

Theorem 2.3.2 (norm). A norm should satisfy

$$\|A\| \ge 0$$

$$\|A + B\| \le \|A\| + \|B\| \tag{2.3.4}$$

$$\|\alpha A\| = |\alpha|\|A\|$$

where α is a scalar.

Proof. Checking the middle one, we have

$$\|A + B\| = \max_{\|x\|=1} \|(A + B)x\| \le \max_{\|x\|=1} \|Ax\| + \|Bx\| \le \max_{\|x\|=1} \|Ax\| + \max_{\|x\|=1} \|Bx\|$$

The last step is because we can choose different x to maximize $\|Ax\|$ and $\|Bx\|$. ■

Hence, here we have the definition of matrix norm:

Definition 2.3.4 (Matrix Norm). A matrix norm is a function $\|\cdot\|$

$$\|A\|_l \equiv \max_{x\neq 0} \frac{\|Ax\|_l}{\|x\|_l} = \max_{\|x\|_l=1} \|Ax\|_l$$

From the definition, above we have

$$\frac{\|Ax\|}{\|x\|} \leq \|A\|, \quad \forall x$$

so

$$\|Ax\| \leq \|A\|\|x\|$$

Note. For the relative error,

$$\frac{|\hat{\alpha}-\alpha|}{|\alpha|}$$

but in practice, α is usually unknown, so we can only use

$$\frac{|\hat{\alpha}-\alpha|}{|\hat{\alpha}|}$$

If

$$\frac{|\hat{\alpha}-\alpha|}{|\hat{\alpha}|} \leq 0.1$$

then

$$\frac{1}{1.1}\frac{|\hat{\alpha}-\alpha|}{|\hat{\alpha}|} \leq \frac{|\hat{\alpha}-\alpha|}{|\alpha|} \leq \frac{1}{0.9}\frac{|\hat{\alpha}-\alpha|}{|\hat{\alpha}|}$$

By

$$|\alpha| - |\hat{\alpha}| \leq |\hat{\alpha}-\alpha| \leq 0.1|\hat{\alpha}|$$

$$|\alpha| \leq 1.1|\hat{\alpha}|$$

Similarly,

$$|\hat{\alpha}| - |\alpha| \leq 0.1|\hat{\alpha}|$$

$$|\alpha| \geq 0.9|\hat{\alpha}|$$

2.3.2 Condition of a Linear System

Solving a Linear System

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}, \text{ solution } = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Now we slightly modify the right-hand side to

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32.1 \\ 22.9 \\ 33.1 \\ 30.9 \end{pmatrix}, \text{ solution } = \begin{pmatrix} 9.2 \\ -12.6 \\ 4.5 \\ -1.1 \end{pmatrix}$$

We can see that a small change can lead to a large error. Or if we slightly modify the coefficient matrix

$$\begin{pmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}, \text{ solution } = \begin{pmatrix} -81 \\ 137 \\ -34 \\ 22 \end{pmatrix}$$

which is even worse.
Now, we consider the gereral case

$$Ax = b$$

1° First, we slightly modify right-hand side

$$A(x + \delta x) = b + \delta b$$

then we get

$$\delta x = A^{-1}\delta b \quad \Rightarrow \quad \|\delta x\| \leq \|A^{-1}\|\|\delta b\|$$

and we have

$$\|b\| \leq \|A\|\|x\|$$

so

$$\boxed{\frac{\|\delta x\|}{\|x\|} \leq \|A^{-1}\|\|A\|\frac{\|\delta b\|}{\|b\|}}$$

2° Next, we slightly modify the coefficient matrix

$$(A + \delta A)(x + \delta x) = b$$

$$Ax + A\delta x + \delta A(x + \delta x) = b$$

then we have

$$\delta x = -A^{-1}\delta A(x + \delta x)$$

imply the norm

$$\|\delta x\| \leq \|A^{-1}\|\|\delta A\|(\|x\| + \|\delta x\|)$$

so

$$\boxed{\frac{\|\delta x\|}{\|x + \delta x\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta A\|}{\|A\|}}$$

> **Note.** We here estimate only
>
> $$\frac{\|\delta x\|}{\|x + \delta x\|} \text{ instead of } \frac{\|\delta x\|}{\|x\|};$$
>
> since the disscussion earlier.

Clearly, the error is related to $\|A\|\|A^{-1}\|$, which is called the condition number of A, a smaller condition number means a more stable system.

Chapter 3

BLAS

Lecture 5

2026.03.21

3.1 BLAS basic

3.1.1 BLAS: Basic Linear Algebra Subroutines

Usually we just use 90% of the time in 10% of the code. So, accelerating the 10% of the code can significantly improve the overall performance. In linear algebra, we use BLAS (Basic Linear Algebra Subroutines) to accelerate the performance of the code, which is a kind of "software" face of optimization.

> **Note.** For GPU, TPU, and other hardware, those are the "hardware" face of optimization.

For example,

```
daxpy(n, alpha, p, inc, w, inc);
daxpy(n, malpha, q, inc, r, inc);

rtr = ddot(n, r, inc, r, inc);
rnorm = sqrt(rtr);
tnorm = sqrt(ddot(n, t, inc, t, inc));
```

- `ddot`: inner product
- `daxpy`: ax + y, x, y are vectors and a is a scalar

> **Note.** Fortran is the first high level programming language, and it is still widely used in scientific computing. The BLAS library is come up before the C language.

3.1.2 Level 1 BLAS

The level 1 BLAS includes:

- dot product
- constant times a vector plus a vector
- rotation (don ' t need to know what this is now)
- copy vector x to vector y

- swap vector x and vector y
- length of a vector ($\sqrt{x_1^2 + \cdots + x_n^2}$)
- sum of absolute values ($|x_1| + \cdots + |x_n|$)
- constant times a vector
- index of element having maximum absolute value

These are all $O(n)$ operations.

```
dw = ddot(n, dx, incx, dy, incy)
```

$$w = \sum_{i=1}^{n} dx_{1+(i-1)\text{incx}} dy_{1+(i-1)\text{incy}}$$

```
dw = ddot(n, dx, 2, dy, 2)
```

$$w = dx_1 dy_1 + dx_3 dy_3 + \cdots$$

> **Note.** `sdot` is single precision, `ddot` is for double precision.

If we want $y = ax + y$

```
daxpy(n, da, dx, incx, dy, incy)
```

For reseacher in this field, it is hard to decide which operations can be put into BLAS. Further, there is another problem: Originally, BLAS is designed for Fortran, but now we have many programming languages. For C calling Fortran

```
rtr = ddot_(&n, r, &inc, r, &inc);
```

- `ddot_`: calling Fortran subroutines (machine dependent)
- `&n`: call by reference for Fortran subroutine.
- Array index starts from 1 in Fortran, but starts from 0 in C. But this should not cause problems here.

There is a CBLAS interface for C, i.e. C call C library,

```
rtr = ddot(n, r, inc, r, inc);
```

We don't need to handle the issue.
Nowadays, we have BLAS libraries available on most computers, for a Linux system

```
$ ls /usr/lib| grep blas
libblas
libblas.a
libblas.so
libblas.so.3
libblas.so.3gf
libcblas.so.3
libcblas.so.3gf
libf77blas.so.3
libf77blas.so.3gf
```

- .a: static library
- .so: dynamic library
- .3: versus
- .3gf: g77 versus gfortran

3.1.3 Level 2 BLAS

Level 2 BLAS involves $O(n^2)$ operations, where n is the size of matrices.

Example. For a Matrix-vector product:

$$Ax : (Ax)_i = \sum_{j=1}^{n} A_{ij}x_j,\ i = 1, \ldots, m$$

This kind of operation using level 1 BLAS will be too slow

$$y = \alpha Ax + \beta y, \quad y = \alpha A^T x + \beta y, \quad y = \alpha \bar{A}^T x + \beta y$$

α, β are scalars, x, y are vectors, A is a matrix, and $\bar{A}^T$ is the conjugate transpose (or Hermitian transpose) of A. For real matrices,

$$\bar{A}^T = A^T$$

Lower- or upper-triangular matrix-vector products

$$x = Tx, x = T^T x, x = \bar{T}^T x$$

x is a vector and T is a lower or upper triangular matrix.

Example. Rank-one and rank-two updates

$$A = \alpha x y^T + A, H = \alpha x \bar{y}^T + \bar{\alpha} y \bar{x}^T + H$$

- H is a Hermitian matrix ($H = \bar{H}^T$, symmetric for real numbers)
- rank: # of independent rows (columns) of a matrix.

xy^T is a rank one matrix. For example,

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 3 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ 6 & 8 \end{pmatrix}$$

This operation need $O(n^2)$ operations.

$xy^T + yx^T$ is a rank-two matrix

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 3 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ 6 & 8 \end{pmatrix}, \begin{pmatrix} 3 \\ 4 \end{pmatrix} \begin{pmatrix} 1 & 2 \end{pmatrix} = \begin{pmatrix} 3 & 6 \\ 4 & 8 \end{pmatrix}$$

Example. Solution of triangular equations

$x = T^{-1}y$

$$\begin{pmatrix} T_{11} & & & \\ T_{21} & T_{22} & & \\ \vdots & \vdots & \ddots & \\ T_{n1} & T_{n2} & \cdots & T_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

The solution is

$$\begin{aligned} x_1 &= y_1/T_{11} \\ x_2 &= (y_2 - T_{21}x_1)/T_{22} \\ x_3 &= (y_3 - T_{31}x_1 - T_{32}x_2)/T_{33} \end{aligned}$$

Number of multiplications/divisions:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

3.1.4 Level 3 BLAS

Example. Matrix-matrix product

$$C = \alpha AB + \beta C, \quad C = \alpha A^T B + \beta C,$$
$$C = \alpha AB^T + \beta C, \quad C = \alpha A^T B^T + \beta C$$

Example. Rank-k and rank-$2k$ updates.

Example. Multiplying a matrix by a triangular matrix

$$B = \alpha TB$$

Example. Solving triangular systems of equations with multiple right-hand side:

$$B = \alpha T^{-1}B$$

BLAS does not include subroutines for solving general linear systems or eigenvalues. Because these problems are not as common as the above operations.

3.2 Optimized BLAS

3.2.1 Memory Management

Here is a C program

```c
#define n 3000
double a[n][n], b[n][n], c[n][n];

int main()
{
   int i, j, k;
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         a[i][j] = 1; b[i][j] = 1;
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         c[i][j] = 0;
         for (k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
      }
}
```

```
$ gcc -O3 mat.c; time ./a.out
real    1m24.909s
user    1m24.534s
sys     0m0.193s
```

We now test in the MATLAB environment

```
$ matlab -singleCompThread
>> n = 3000;
>> A = randn(n,n); B = randn(n,n);
>> tic; C = A*B; toc
Elapsed time is 1.708523 seconds.
```

An issue about timing is elapsed time versus CPU time

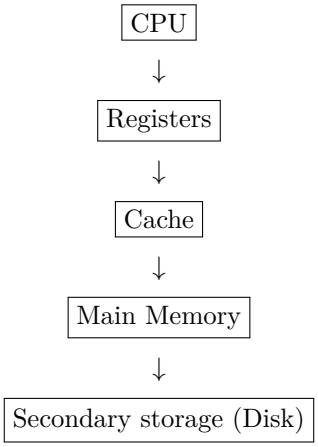
```
>> A = randn(n,n); B = randn(n,n);
>> t = cputime; C = A*B; t = cputime -t
t = 1.3000
```

They are similar if no other jobs are running on this machine.
Results of using multi-threading (the default of MATLAB)

```
$ matlab
>> n = 3000;
>> A = randn(n,n); B = randn(n,n);
>> tic; C = A*B; toc
```

```
Elapsed time is 0.426942 seconds.
>> A = randn(n,n); B = randn(n,n);
>> t = cputime; C = A*B; t = cputime -t
t = 5.1200
```

The CPU time here is the total time used by all threads.
Optimized BLAS is an implementation that takes the advantage of memory hierarchies



Note. In the reality, the architecture of CPU is more complicated, and there are multiple levels of cache.

Definition 3.2.1 (Block algorithm). **Block algorithms** is a way to transfer sub-matrices between different levels of storage. They localize operations to achieve good performance.

Definition 3.2.2 (Page fault). **Page fault** is an event where an operand is not available in main memory and must be transported from secondary memory. Usually if a page is moved to the main memory, it overwrites page least recently used.

Example. $C = AB + C$, $n = 1,024$

Here is the assumption:

- a page 65,536 doubles = 64 columns
- 16 pages for each matrix
- 48 pages for three matrices
- available memory 16 pages
- matrices access: column oriented

Note.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

The access way of the elements:
column oriented: 1 3 2 4

- row oriented: 1 2 3 4

Access each row of A: 16 page faults, $1024/64 = 16$

Example. Here is approch 1:

```
for i =1:n
  for j=1:n
    for k=1:n
      c(i,j) = a(i,k)*b(k,j)+c(i,j);
    end
  end
end
```

At each `(i,j)`: each row `a(i, 1:n)` causes 16 page faults. Total:

$$1024^2 \times 16$$

at least **16 million page faults**.

Example. Here is approach 2:

```
for j=1:n
  for k=1:n
    for i=1:n
      c(i,j) = a(i,k)*b(k,j)+c(i,j);
    end
  end
end
```

For each j, access all columns of A. A needs 16 pages, but B and C take spaces as well. So A must be read for every j. For each j, 16 page faults for A.

$$1024 \times 16$$

and C, B : 16 page faults. This approach has a dramatic improvement, which is **16,384 page faults**. If we implement back to C,

```
#define n 3000
double a[n][n], b[n][n], c[n][n];

int main()
{
   int i, j, k;
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         a[i][j] = 1; b[i][j] = 1;
         c[i][j] = 0;
      }

   for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++)
```

```
        for (i = 0; i < n; i++)
           c[i][j] += a[i][k]*b[k][j];
   }
}
```

```
$ gcc -O3 mat1.c; time ./a.out
real    4m20.247s
user    4m19.761s
sys     0m0.154s
```

The performance is much worse than the previous one. The reason is that C language is row oriented.

Example. Here is approach 3 (block algorithm with $nb = 256$):

```
for j=1:nb:n
  for k=1:nb:n
    for jj=j:j+nb-1
      for kk=k:k+nb-1
        c(:,jj) = a(:,kk)*b(kk,jj)+c(:,jj);
      end
    end
  end
end
```

In MATLAB, `1:256:1025` means 1, 257, 513, 769. Note that we calculate

$$\begin{pmatrix} A_{11} & \cdots & A_{14} \\ & \vdots & \\ A_{41} & \cdots & A_{44} \end{pmatrix} \begin{pmatrix} B_{11} & \cdots & B_{14} \\ & \vdots & \\ B_{41} & \cdots & B_{44} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + \cdots + A_{14}B_{41} & \cdots \\ \vdots & \ddots \end{pmatrix}$$

We split the original matrix into 16 blocks, for each block 256×256

$$\begin{aligned} C_{11} &= A_{11}B_{11} + \cdots + A_{14}B_{41} \\ C_{21} &= A_{21}B_{11} + \cdots + A_{24}B_{41} \\ C_{31} &= A_{31}B_{11} + \cdots + A_{34}B_{41} \\ C_{41} &= A_{41}B_{11} + \cdots + A_{44}B_{41} \end{aligned}$$

For each (j, k), $B_{k,j}$ is used to add $A_{:,k}B_{k,j}$ to $C_{:,j}$. For instance:

$$\begin{aligned} C_{11} &\leftarrow C_{11} + A_{11}B_{11} \\ &\vdots \\ C_{41} &\leftarrow C_{41} + A_{41}B_{11} \end{aligned}$$

Then use Approach 2 for $A_{:,1}B_{11}$.

Remark (Analysis)**.** Here is the analysis of page faults for this approach:

- $A_{:,1}$: 256 columns, $1024 \times 256/65536 = 4$ pages.
- $A_{:,1}, \ldots, A_{:,4}$: $4 \times 4 = 16$ page faults in calculating $C_{:,1}$

- For A: 16×4 page faults
- B: 16 page faults
- C: 16 page faults

3.2.2 LAPACK

BLAS defines only operations such as matrix-matrix products. LAPACK –Linear Algebra PACKage, based on BLAS, which can solve

- Systems of linear equations
- Least-squares solutions of linear systems of equations
- Eigenvalue problems
- Singular value problems

Subroutines in LAPACK classified as three levels:

1. **Driver routines**: Subroutines that solve complete problems, such as linear systems, eigenvalue problems, and singular value problems.
2. **Computational routines**: Subroutines that perform a specific computation such as LU factorization, QR factorization, and Hessenberg reduction.
3. **Auxiliary routines**: Subroutines that perform auxiliary computations, which is kind of extension of BLAS.

Definition 3.2.3 (Name of LAPACK Driver/Computational Routines). Name of LAPACK Driver and Computational Routines has the form of

`XYYZZZ`

- `X`: data type (S: single, D: double, C: complex, Z: complex double)
- `YY`: the type of matrix
  - `GB`: general banded
  - `GE`: general (i.e. unsymmetric, in some cases rectangular)

  Note. Banded matrix: a band of nonzeros along diagonals

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

- `ZZZ`: the type of computation performed
  - `SV`: simple driver of solving general linear systems.
  - `TRF`: Factorization.
  - `TRS`: Use the factorization to solve $Ax = b$ by forward or backward substitution.

– `CON`: Estimate the reciprocal of the condition number.

Example. `SGESV`: Simple driver of solving single general linear systems.

Example. `SGBSV`: Simple driver of solving single general banded linear systems.

For the block algorithm in LAPACK, LU factorization `DGETRF` takes $O(n^3)$ operations, speed in megaflops (MFLOPS) is defined as

$$\text{MFLOPS} = \frac{\text{number of floating point operations}}{\text{time in seconds} \times 10^6}$$

	No. of CPUs	Block size	100	1000
Dec Alpha Miata	1	28	172	370
Compaq AlphaServer DS-20	1	28	353	440
IBM Power 3	1	32	278	551
IBM PowerPC	1	52	77	148
Intel Pentium II	1	40	132	250
Intel Pentium III	1	40	143	297
SGI Origin 2000	1	64	228	452
SGI Origin 2000	4	64	190	699
Sun Ultra 2	1	64	121	240
Sun Enterprise 450	1	64	163	334

Table 3.1: Performance of block LU factorization `DGETRF` in LAPACK on different machines.

We can see that block algorithms are not very effective for small-sized problems. In the Intel Pentium III, the CPU clock time is 550 MHz, so the theoretical peak performance is just almost achieved for $n = 1000$, which means that the block algorithm is very effective for large-sized problems.

Note (ATLAS). There is also some software packages for optimized BLAS, such as ATLAS (Automatically Tuned Linear Algebra Software), which can automatically tune the parameters of the block algorithm to achieve better performance. Which means compiled for your architecture, things related to your CPU, size of cache, RAM.

Chapter 4

Iterative Methods for Linear Systems

4.1 Matrix Storage

Lecture 6

4.1.1 Sparse Matrix Storage

2026.04.20

Definition 4.1.1 (Sparse Matrix)**.** A sparse matrix is a matrix in which most of the elements are zero.

If we can store only the non-zero elements of a sparse matrix, we can save a lot of memory and to handle a large linear system efficiently.

Example.

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 3 & 4 & 0 & 5 \\ 6 & 0 & 7 & 8 \\ 0 & 0 & 10 & 11 \end{pmatrix}$$

Here is the easiest scheme to store the non-zero elements of a sparse matrix **coordinate format** (COO):

```
a         1 3 6 4 7 10 2 5 8 11
arow_ind  1 2 3 2 3 4  1 2 3  4
acol_ind  1 1 1 2 3 3  4 4 4  4
```

However, indices may not be well ordered, when doing matrix addition $A + B$, it may be hard to find the corresponding elements in A and B.
But for $y = Ax$, we can simply loop through the non-zero elements

```
for l = 1:nnz % nnz is the number of non-zero elements
    i = arow_ind(l)
    j = acol_ind(l)
    y(i) = y(i) + a(l) * x(j)
end
```

x is a vector in dense format

Note. In Deep Learning Inference, it is a kind of dense dense matrix multiplication, but nowadays, scientists tried to sparsify the matrix. For example, "Quantization" is a technique to reduce the precision of the weights, which can lead to sparsity in the matrix. Then we can use sparse matrix storage and sparse matrix multiplication to speed up the inference process.

Example. For another problem, we now try to get a "column" of the matrix.

When we need to get the i-th column of the matrix, for example, to solve $Lx = b$

$$\begin{pmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & & \ddots & \\ l_{n1} & l_{n2} & & l_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

once we get the L we have to do the substraction, which needs to get the i-th column of L

$$\begin{pmatrix} b_2 \\ \vdots \\ b_n \end{pmatrix} - x_1 \begin{pmatrix} l_{21} \\ \vdots \\ l_{n1} \end{pmatrix}$$

```
for l = 1:nnz
    if acol_ind(l) == i
        x(arow_ind(l)) = a(l)
    end
end
```

This is not efficient, because the complexity is $O(nnz)$, which is the size of the non-zero elements, which can be large.
Now we introduce another scheme to store the non-zero elements of a sparse matrix **Compressed Column Format**:

```
a        1 3 6 4 7 10 2 5 8 11
arow_ind 1 2 3 2 3 4  1 2 3  4
acol_ptr 1 4 5 7 11
```

To access j-th column, we can simply get

from `acol_ptr(j)` to `acol_ptr(j+1)-1`

In this format, we can get $A + B$ easily by

```
for j = 1:n
    get A's j-th column
    get B's j-th column
    vector addition
end
```

In this way, C will still in CSR format.
Now consider

$$y = Ax = A_{:,1}x_1 + A_{:,2}x_2 + \cdots + A_{:,n}x_n$$

we can loop through the columns of A

```
for j = 1:n
    for l = acol_ptr(j):acol_ptr(j+1)-1
        i = arow_ind(l)
        y(i) = y(i) + a(l) * x(j)
    end
end
```

> **Note.** In the real implementation (e.g. C/C++), there is a "pointers" array to store the starting index of each row in the data array, which is also a kind of CCF format. But using three arrays is more intuitive and look at the history of the development of programming languages, pointers are not design for this simple situation.

> **Remark.** The row index might not be well ordered in the same column in CCF.

However, accessing the i-th row is not efficient in CCF. We can also use **Compressed Row Format** (CRF) to store the non-zero elements of a sparse matrix.

```
a        1 2 3 4 5 6 7 8 10 11
acol_ind 1 4 1 2 4 1 3 4 3  4
arow_ptr 1 3 6 9 11
```

In the C implementation, the index starts from 0, the problem might happen when you call a Fortran library in C.

```
a        1 3 6 4 7 10 2 5 8 11
arow_ind 0 1 2 1 2 3  0 1 2  3
acol_ptr 0 3 4 6 10
```

4.1.2 Factorization and Fill-in

> **Definition 4.1.2** (Fill-in). Fill-in is the phenomenon that zero elements in the original matrix become non-zero elements in the factorized matrices (e.g. L and U in LU factorization).

Consider the following program

```
A = sprandsym(200, 0.05, 0.01, 1) ;
L = chol(A)' ;
spy(A) ;
print -deps A
spy(L) ;
print -deps L
```

- 0.05: density
- 0.01: 1/(condition number)
- 1: type of matrix, 1 gives a matrix with
- 1/(condition number) exactly 0.01

`spy`: draw the sparsity pattern

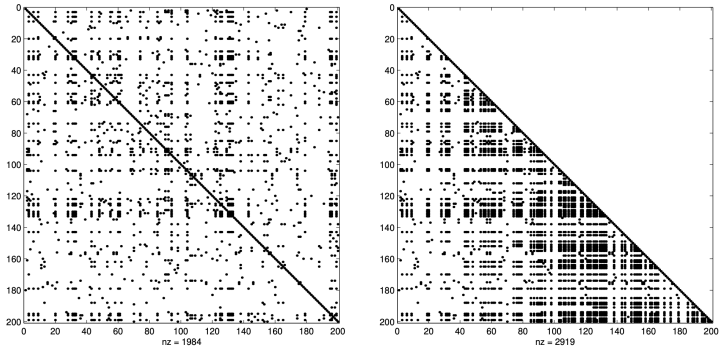


Figure 4.1: Sparsity pattern of A

Clearly L is denser than A.

Example.

$$A = \begin{pmatrix} 3 & 2 & 1 & 2 \\ 2 & 4 & 0 & 0 \\ 1 & 0 & 5 & 0 \\ 2 & 0 & 0 & 6 \end{pmatrix}$$

We will get a very dense L if we do Cholesky factorization on A.

$$\text{chol}(A) = \begin{pmatrix} 1.7321 & 0 & 0 & 0 \\ 1.1547 & 1.6330 & 0 & 0 \\ 0.5774 & -0.4082 & 2.1213 & 0 \\ 1.1547 & -0.8165 & -0.4714 & 1.9437 \end{pmatrix}$$

If we do the reordering

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \quad AP^T = \begin{pmatrix} 2 & 1 & 2 & 3 \\ 0 & 0 & 4 & 2 \\ 0 & 5 & 0 & 1 \\ 6 & 0 & 0 & 2 \end{pmatrix}$$

$$PAP^T = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 2 & 1 & 2 & 3 \\ 0 & 0 & 4 & 2 \\ 0 & 5 & 0 & 1 \\ 6 & 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 6 & 0 & 0 & 2 \\ 0 & 5 & 0 & 1 \\ 0 & 0 & 4 & 2 \\ 2 & 1 & 2 & 3 \end{pmatrix}$$

Then we get

$$\text{chol}(PAP^T) = \begin{pmatrix} 2.4495 & 0 & 0 & 0 \\ 0 & 2.2361 & 0 & 0 \\ 0 & 0 & 2.0000 & 0 \\ 0.8165 & 0.4472 & 1.0000 & 1.0646 \end{pmatrix}$$

When we done the reordering, the $\text{chol}(PAP^T)$ is much sparser. Next we solve the original system $Ax = b$ by

$$(PAP^T)Px = Pb$$

Get Px first, then get x.
The MATLAB provides method such as

- Column Count Reordering

- Reverse Cuthill-McKee Reordering
- Minimum Degree Reordering
- Nested Dissection Permutation

> **Remark.** In pratice, minimum fill-in problem is NP-hard, which is a combinatorial optimization problem, we can only get an approximation solution by some heuristics.

However, minimum fill-in may not be the best: we need to consider the numerical stability, implementation efforts. Maybe finding a good reordering is more complicated than just doing the factorization and solve the system.

Lecture 7

2026.04.27

4.2 Iterative Methods

An iterative method is to have the process of

$$x_1, x_2, \ldots, \to x^*$$

We hope to find

$$Ax^* = b$$

This type of method should be faster than Gaussian elimination $O(n^3)$. If $x_k \to x_{k+1}$ takes $O(n^r)$, then we can have $O(ln^r)$, where l is the number of iterations. There are two properties should be satisfied for an iterative method:

- Numerical stability
- Sparsity (Fill-in)

4.2.1 Jacobi Method

> **Example.** Consider the system of equations $Ax = b$

We have

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{aligned}$$

The Jacobi method is to update x_k to x_{k+1} by

$$\begin{aligned} (x_{k+1})_1 &= (b_1 - a_{12}(x_k)_2 - a_{13}(x_k)_3)/a_{11} \\ (x_{k+1})_2 &= (b_2 - a_{21}(x_k)_1 - a_{23}(x_k)_3)/a_{22} \\ (x_{k+1})_3 &= (b_3 - a_{31}(x_k)_1 - a_{32}(x_k)_2)/a_{33} \end{aligned}$$

In the general case, we should do the procedure of

JACOBI$(A, b, x^{(0)})$
1 **for** $i = 1 \ldots n$
2 $\quad (x_{k+1})_i = (b_i - \sum_{j=1}^{i-1} a_{ij}(x_k)_j - \sum_{j=i+1}^{n} a_{ij}(x_k)_j)/a_{ii}$

Definition 4.2.1 (Jacobi Method). The Jacobi method is to update x_k to x_{k+1} by

$$(x_{k+1})_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}(x_k)_j - \sum_{j=i+1}^{n} a_{ij}(x_k)_j\right)$$

4.2.2 Gauss-Seidel Method

Now consider Gauss-Seidel method, which is to update x_k to x_{k+1} using the recently updated values. We have

$$\begin{aligned}(x_{k+1})_1 &= (b_1 - a_{12}(x_k)_2 - a_{13}(x_k)_3)/a_{11}\\ (x_{k+1})_2 &= (b_2 - a_{21}(x_{k+1})_1 - a_{23}(x_k)_3)/a_{22}\\ (x_{k+1})_3 &= (b_3 - a_{31}(x_{k+1})_1 - a_{32}(x_{k+1})_2)/a_{33}\end{aligned}$$

For the general case, we should do the procedure of

GAUSS-SEIDEL$(A, b, x^{(0)})$
1 **for** $i = 1..n$
2 $\quad (x_{k+1})_i = (b_i - \sum_{j=1}^{i-1} a_{ij}(x_{k+1})_j - \sum_{j=i+1}^{n} a_{ij}(x_k)_j)/a_{ii}$

Definition 4.2.2. The Gauss-Seidel method is to update x_k to x_{k+1} by

$$(x_{k+1})_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}(x_{k+1})_j - \sum_{j=i+1}^{n} a_{ij}(x_k)_j\right)$$

The iterative method can be divergent,

Example.

$$A = \begin{pmatrix}1 & 2 & 2\\ 2 & 1 & 2\\ 2 & 2 & 1\end{pmatrix}, b = \begin{pmatrix}5\\5\\5\end{pmatrix}, \text{sol} = \begin{pmatrix}1\\1\\1\end{pmatrix}$$

The determinant of A is -9, so A is invertible. However, the iterative method will diverge

$$x_0 = \begin{pmatrix}0\\0\\0\end{pmatrix},\ x_1 = \begin{pmatrix}5\\5\\5\end{pmatrix},\ x_2 = \begin{pmatrix}5-10-10\\5-10-10\\5-10-10\end{pmatrix} = \begin{pmatrix}-15\\-15\\-15\end{pmatrix},\ x_3 = \begin{pmatrix}5+30+30\\5+30+30\\5+30+30\end{pmatrix} = \begin{pmatrix}65\\65\\65\end{pmatrix}$$

Here is the convergence analysis of the iterative method. The goal is to find when the iterative method will get the solution of $Ax = b$.

$$\begin{aligned}a_{11}x_1 &= b_1 - a_{12}x_2 - a_{13}x_3\\ a_{22}x_2 &= b_2 - a_{21}x_1 - a_{23}x_3\\ a_{33}x_3 &= b_3 - a_{31}x_1 - a_{32}x_2\end{aligned}$$

Hence, we can rewrite the above equations as

$$\begin{pmatrix}a_{11} & & \\ & a_{22} & \\ & & a_{33}\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} = \begin{pmatrix}0 & -a_{12} & -a_{13}\\ -a_{21} & 0 & -a_{23}\\ -a_{31} & -a_{32} & 0\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} + b$$

If

$$M = \begin{pmatrix} a_{11} & & \\ & a_{22} & \\ & & a_{33} \end{pmatrix}, N = \begin{pmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{pmatrix}$$

then

$$A = M - N, \quad Mx_{k+1} = Nx_k + b$$

Here we define the spectral radius

Definition 4.2.3 (Spectral Radius)**.** The spectral radius of a matrix A is

$$\rho(A) = \max\{|\lambda| : \lambda \in \lambda(A)\}$$

where $\lambda(A)$ is the set of eigenvalues of A.

Then we have the following theorem

Theorem 4.2.1. Assume

$$A = M - N$$

where

$$A, M \text{are non-singular and } \rho(M^{-1}N) < 1$$

Then

$$Mx_{k+1} = Nx_k + b$$

leads to convergence of $\{x_k\}$ to $A^{-1}b$ for any initial guess x_1.

Proof. We have

$$Ax = b \Rightarrow Mx = Nx + b$$

Get the iterative method in

$$M(x_{k+1} - x) = N(x_k - x)$$

Hence,

$$x_{k+1} - x = M^{-1}N(x_k - x)$$

Get the general formula

$$x_{k+1} - x = (M^{-1}N)^k(x_1 - x)$$

With the condition $\rho(M^{-1}N) < 1$, we have

$$\lim_{k\to\infty} (M^{-1}N)^k = 0$$

Note. The reason of condition

$$\rho(M^{-1}N) < 1 \Rightarrow \lim_{k\to\infty} (M^{-1}N)^k = 0$$

is quite complicated, we omit the proof here.

Hence,

$$\lim_{k\to\infty} x_{k+1} - x = 0$$

■

4.2.3 Theoretical Explanation of Gauss-Seidel Method

Now we try to give Gauss-Seidel method a theoretical explanation. We construct the following opitmization problem

$$\min_x f(x) = \frac{1}{2}x^T Ax - b^T x$$

which is the same as solving $Ax = b$ if A is a symmetric positive definite. We let

$$f(x) = \frac{1}{2}x^T Ax - b^T x$$

then

$$\nabla f(x) \equiv \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} = Ax - b$$

The derivation will be

$$\begin{aligned} x^T Ax &= \sum_{i=1}^{n} x_i (Ax)_i \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i a_{ij} x_j \\ &= \underbrace{x_1 A_{11}x_1 + \cdots + x_1 A_{1n}x_n}_{\text{first row}} + \underbrace{x_2 A_{21}x_1 + \cdots + x_2 A_{2n}x_n}_{\text{second row}} + \cdots + \underbrace{x_n A_{n1}x_1 + \cdots + x_n A_{nn}x_n}_{n\text{-th row}} \end{aligned}$$

Therefore,

$$\begin{aligned} \frac{\partial x^T Ax}{\partial x_1} &= (2A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n) + x_2 A_{21} + \cdots + x_n A_{n1} \\ &= 2(A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n) \qquad \text{(since } A \text{ is symmetric)} \end{aligned}$$

> **Note.** Coordinate-wise minimization is a classic way for solving optimization problems. Sequentially optimizing one variable at a time
>
> $$\begin{aligned} \min_{x_1} f(x_1, x_2^k, \ldots, x_n^k) \\ \min_{x_2} f(x_1^{k+1}, x_2, \ldots, x_n^k) \\ \min_{x_3} f(x_1^{k+1}, x_2^{k+1}, x_3, \ldots, x_n^k) \\ \vdots \end{aligned}$$

Assume x_i is updated

$$\min_d \frac{1}{2}(x + de_i)^T A(x + de_i) - b^T(x + de_i)$$

Next,

$$\begin{aligned} \min_d \frac{1}{2}d^2 A_{ii} + d(Ax)_i - db_i \\ A_{ii}d + (Ax)_i - b_i = 0 \end{aligned}$$

The minimum is achieved at

$$d = \frac{b_i - (Ax)_i}{A_{ii}}$$

So,

$$x_i + d \leftarrow \frac{b_i - \sum_{j:j\neq i} A_{ij}x_j}{A_{ii}}$$

We can get the Gauss-Seidel update rule.

4.3 Conjugate Gradient Method

This method can use only symmetric positive definite matrices. We first disscuss the **steepest descent method**.

4.3.1 Steepest Descent Method

We construct the following optimization problem

$$\min_x f(x) = \frac{1}{2}x^T Ax - b^T x$$

Using taylor expansion, we have

$$f(x) = f(x_k) + \nabla f(x_k)^T(x - x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k) + \cdots$$

In this method, we omit the higher order terms and only consider the first two terms. Hence, we have

$$f(x) \approx f(x_k) + \nabla f(x_k)^T(x - x_k)$$

The first term $f(x_k)$ is a constant, so we only need to minimize $\nabla f(x_k)^T(x - x_k)$. If

$$\min_{\|x - x_k\| = 1} \nabla f(x_k)^T(x - x_k)$$

then

$$x - x_k = -\frac{\nabla f(x_k)}{\|\nabla f(x_k)\|}$$

is the direction of steepest descent, which is the unit vector of $-\nabla f(x_k)$.

> **Note.** We need to
>
> $$\|x - x_k\| = 1$$
>
> otherwise, we can make x go to infinity and make $\nabla f(x_k)^T(x - x_k)$ go to negative infinity.

Now we have

$$\nabla f(x_k) = Ax_k - b$$

Let

$$r = -\nabla f(x_k) = b - Ax_k, \quad x = x_k$$

We minimize along the direction of r

$$\min_\alpha \frac{1}{2}(x + \alpha r)^T A(x + \alpha r) - (x + \alpha r)^T b$$
$$\min_\alpha \frac{1}{2}\alpha^2 r^T Ar + \alpha r^T Ax - \alpha r^T b$$
$$\min_\alpha \frac{1}{2}\alpha^2 r^T Ar + \alpha r^T (Ax - b)$$

Then, it becomes an one-dimensional optimization problem

$$\alpha r^T Ar + r^T(Ax - b) = 0$$
$$\alpha = \frac{r^T(b - Ax)}{r^T Ar}$$

Note. $r^TAr \neq 0$ since A is positive definite and $r \neq 0$.

Now $r = b - Ax$, we have

$$\alpha = \frac{(b-Ax)^T(b-Ax)}{(b-Ax)^TA(b-Ax)} = \frac{r^Tr}{r^TAr}$$

Hence, we can get the update rule of the steepest descent method

SteepestDescent(A, b)
1 $k = 0,\ x_0 = 0,\ r_0 = b$
2 **while** $r_k \neq 0$
3 $\quad k = k + 1$
4 $\quad \alpha_k = \frac{r_{k-1}^T r_{k-1}}{r_{k-1}^T A r_{k-1}}$
5 $\quad x_k \leftarrow x_{k-1} + \alpha_k r_{k-1}$
6 $\quad r_k \leftarrow b - Ax_k$

However, the steepest descent method may converge very slowly, becasue r is not a good direction to go. So we have to construct a better direction. Before that, we first construct the general search direction method.

Definition 4.3.1 (Search Direction)**.** Suppose we have x_k obtained by

$$\min_{\alpha} f(x_{k-1} + \alpha p_k)$$

where p_k is the search direction.

Then,

$$\begin{aligned} f(x_{k-1} + \alpha p_k) &= \frac{1}{2}(x_{k-1} + \alpha p_k)^T A(x_{k-1} + \alpha p_k) - b^T(x_{k-1} + \alpha p_k) \\ &= C + \alpha p_k^T(Ax_{k-1} - b) + \frac{1}{2}\alpha^2 p_k^T A p_k \end{aligned}$$

So,

$$\alpha = -\frac{p_k^T(r_{k-1})}{p_k^T A p_k}$$

Hence, we have the following gereral search direction method

SearchDirection(A, b)
1 $k = 0,\ x_0 = 0,\ r_0 = b$
2 **while** $r_k \neq 0$
3 $\quad k = k + 1$
4 $\quad$ Choose a search direction p_k such that $p_k^T r_{k-1} \neq 0$
5 $\quad \alpha_k = -\frac{p_k^T r_{k-1}}{p_k^T A p_k}$
6 $\quad x_k \leftarrow x_{k-1} + \alpha_k p_k$
7 $\quad r_k \leftarrow b - Ax_k$

In this setting,

$$x_k = x_0 + \text{span}\{p_1, \ldots, p_k\}$$

The question become how to choose the search direction p_k?

4.3.2 Conjugate Gradient Method

For selecting the search direction, we hope that $p_1, \ldots, p_k$ are linearly independent and

$$x_k = \arg \min_{x \in x_0 + \text{span}\{p_1, \ldots, p_k\}} f(x) \tag{4.1}$$

With the condition 4.1,

$$\text{span}\{p_1, \ldots, p_n\} = \mathbb{R}^n$$

so

$$x_n = \arg \min_{x \in \mathbb{R}^n} f(x)$$

Then

$$Ax_n = b$$

and the procedure will converge in at most n iterations. To maintain the property 4.1, we need to let

$$x_k = x_0 + P_{k-1} y + \alpha p_k$$

where

$$P_{k-1} = (p_1, \ldots, p_{k-1}), \quad y \in \mathbb{R}^{k-1}, \quad \alpha \in \mathbb{R}$$

From the optimality condition, we have

$$\begin{aligned} f(x_k) &= \frac{1}{2}(x_0 + P_{k-1}y + \alpha p_k)^T A(x_0 + P_{k-1}y + \alpha p_k) - b^T(x_0 + P_{k-1}y + \alpha p_k) \\ &= \frac{1}{2}(x_0 + P_{k-1}y)^T A(x_0 + P_{k-1}y) - b^T(x_0 + P_{k-1}y) + \alpha p_k^T A(x_0 + P_{k-1}y) - b^T(\alpha p_k) + \frac{\alpha^2}{2} p_k^T A p_k \\ &= f(x_0 + P_{k-1}y) + \alpha p_k^T A P_{k-1} y - \alpha p_k^T r_0 + \frac{\alpha^2}{2} p_k^T A p_k \end{aligned}$$

In this way, sloving

$$\min_{x \in x_0 + \text{span}\{p_1, \ldots, p_k\}} f(x) = \min_{y, \alpha} f(x_0 + P_{k-1}y + \alpha p_k)$$

is difficult because the term

$$\alpha p_k^T A P_{k-1} y$$

involves both y and α. To make the problem easier, if

$$p_k \in \text{span}\{Ap_1, \ldots, Ap_{k-1}\}^\perp$$

then

$$p_k^T A P_{k-1} y = 0$$

and

$$\min_{x \in x_0 + \text{span}\{p_1, \ldots, p_k\}} f(x) = \min_{y} f(x_0 + P_{k-1}y) + \min_{\alpha}(-\alpha p_k^T r_0 + \frac{\alpha^2}{2} p_k^T A p_k)$$

Then we have two independent optimization problems. Therefore, we require p_k to be A-conjugate to $p_1, \ldots, p_{k-1}$, i.e.

$$p_k^T A p_j = 0, \quad \forall j < k$$

By induction, we already have

$$x_{k-1} = \arg \min_{y} f(x_0 + P_{k-1}y)$$

The solution of the second optimization problem is

$$\alpha = \frac{p_k^T r_0}{p_k^T A p_k}$$

Because of the A-conjugacy, we have

$$\begin{aligned} p_k^T r_{k-1} &= p_k^T (b - Ax_{k-1}) \\ &= p_k^T (b - A(x_0 + P_{k-1} y_{k-1})) = p_k^T r_0 \end{aligned}$$

We have

$$\alpha_k = \frac{p_k^T r_0}{p_k^T A p_k} = \frac{p_k^T r_{k-1}}{p_k^T A p_k}$$

and

$$x_k = x_{k-1} + \alpha_k p_k$$

We can get the following algorithm for the conjugate gradient method

ConjugateGradient(A, b)

1 $k = 0,\ x_0 = 0,\ r_0 = b$
2 **while** $r_k \neq 0$
3 $\quad k = k + 1$
4 $\quad$ Choose any $p_k \in \text{span}\{Ap_1, \ldots, Ap_{k-1}\}^\perp$ such that $p_k^T r_{k-1} \neq 0$
5 $\quad \alpha_k = \frac{p_k^T r_{k-1}}{p_k^T A p_k}$
6 $\quad x_k \leftarrow x_{k-1} + \alpha_k p_k$
7 $\quad r_k \leftarrow b - Ax_k$

We still not pick p_k yet, one common way is to minimize the distance between p_k and r_{k-1}.

> **Note.** Because r_{k-1} is the direction of steepest descent
>
> $$\nabla f(x_{k-1}) = Ax_{k-1} - b = -r_{k-1}$$

The algorithm is become

Conjugate-Gradient(A, b)

1 $k = 0,\ x_0 = 0,\ r_0 = b$
2 **while** $r_k \neq 0$
3 $\quad k = k + 1$
4 $\quad$ **if** $k = 1$
5 $\quad\quad p_1 = r_0$
6 $\quad$ **else**
7 $\quad\quad$ Let p_k minimize $\|p_k - r_{k-1}\|_2$ overall vectors in $p \in \text{span}\{Ap_1, \ldots, Ap_{k-1}\}^\perp$
8 $\quad \alpha_k = \frac{p_k^T r_{k-1}}{p_k^T A p_k}$
9 $\quad x_k = x_{k-1} + \alpha_k p_k$
10 $\quad r_k = b - Ax_k$

We have the following lemma for choosing p_k.

> **Lemma 4.3.1.** If p_k minimizes $\|p_k - r_{k-1}\|_2$ overall vectors in $p \in \text{span}\{Ap_1, \ldots, Ap_{k-1}\}^\perp$, then
>
> $$p_k = r_{k-1} - AP_{k-1} z_{k-1} \tag{4.2}$$
>
> where z_{k-1} solves
>
> $$\min_z \|r_{k-1} - AP_{k-1} z\|_2 \tag{4.3}$$
>
> $P_k = (p_1, \ldots, p_k)$: an $n \times k$ matrix.

Proof. Let z_{k-1} be the solution of (4.3), then and define

$$p = r_{k-1} - AP_{k-1}z_{k-1}$$

The optimality condition of (4.3) is

$$P_{k-1}^T A^T (r_{k-1} - AP_{k-1}z_{k-1}) = 0$$

Thus

$$p^T A^T p_i = 0, \quad \forall i < k$$

Moreover, p is the orthogonal projection of r_{k-1} onto $S = \text{span}\{Ap_1, \ldots, Ap_{k-1}\}^\perp$, so $p = p_k$ is the solution of

$$\min_{p \in S} \|p - r_{k-1}\|_2$$

■

Note. $\nabla f(x) = 0$ is the optimality condition of the optimization problem

$$f(z) = \|r_{k-1} - AP_{k-1}z\|_2^2$$

$$\nabla_z f(z) = -2(AP_{k-1})^T r_{k-1} + 2(AP_{k-1})^T (AP_{k-1})z = 0$$

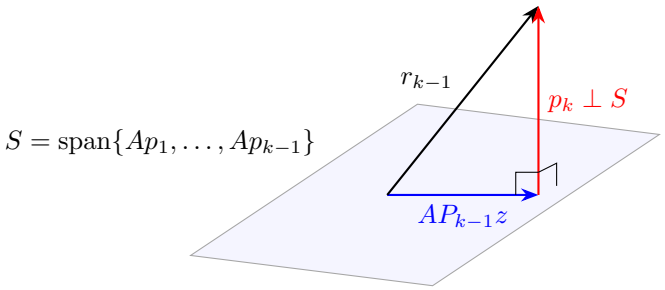


Figure 4.2: The geometric interpretation of the conjugate gradient method.

4.3.3 Properties of Conjugate Gradient Method

Theorem 4.3.1. After j iterations, we have

$$\begin{aligned} r_j &= r_{j-1} - \alpha_j A p_j \\ P_j^T r_j &= 0 \\ \text{span}\{p_1, \ldots, p_j\} &= \text{span}\{r_0, \ldots, r_{j-1}\} \\ &= \text{span}\{b, Ab, \ldots, A^{j-1}b\} \end{aligned} \tag{4.4}$$

$$r_i^T r_j = 0 \text{ for all } i \neq j$$

Proof. We only proof the first property, the others can be proved by induction.

$$\begin{aligned} r_j &= b - Ax_j \\ &= b - A(x_{j-1} + \alpha_{j-1}p_{j-1}) \\ &= r_{j-1} - \alpha_j A p_j \end{aligned}$$

■

From this theorem, we can see that r_i, r_j are orthogonal for $i \neq j$.

Recall. lemma 4.3.1

$$p_k = r_{k-1} - AP_{k-1}z_{k-1}$$

Now we want to find z_{k-1} because A, P_{k-1} are known. We have z_{k-1} is a vector with length $k-1$,

$$z_{k-1} = \begin{pmatrix} \mathbf{w} \\ \mu \end{pmatrix}, \quad \mathbf{w} \in \mathbb{R}^{k-2}, \mu \in \mathbb{R}$$

$$\begin{aligned} p_k &= r_{k-1} - AP_{k-1}z_{k-1} && (4.5) \\ &= r_{k-1} - AP_{k-2}\mathbf{w} - \mu Ap_{k-1} \\ &= (1 + \frac{\mu}{\alpha_{k-1}})r_{k-1} + s_{k-1} && (4.6) \end{aligned}$$

From the thorem, we have

$$r_{k-1} = r_{k-2} - \alpha_{k-1}Ap_{k-1}$$

Definition 4.3.2. We call s_{k-1} the **correction term** of p_k.

$$\begin{aligned} s_{k-1} &= -\frac{\mu}{\alpha_{k-1}}r_{k-1} - AP_{k-2}\mathbf{w} - \mu Ap_{k-1} \\ &= -\frac{\mu}{\alpha_{k-1}}r_{k-2} - AP_{k-2}\mathbf{w} \end{aligned} \tag{4.7}$$

Then, we have

$$r_i^T r_j = 0, \quad \forall i \neq j$$

and from the theorem, we have

$$\begin{aligned} AP_{k-2}\mathbf{w} &\in \text{span}\{Ap_1, \ldots, Ap_{k-2}\} \\ &= \text{span}\{Ab, \ldots, A^{k-2}b\} \\ &\subset \text{span}\{r_0, \ldots, r_{k-2}\} \end{aligned}$$

Hence, $r_{k-1}^T(AP_{k-2}\mathbf{w}) = 0$

$$s_{k-1}^T r_{k-1} = 0 \tag{4.8}$$

Recall. From the earlier results, our goal is to find z_{k-1} such that

$$\|r_{k-1} - AP_{k-1}z\|_2$$

is minimized.

The reason of minimizing $\|r_{k-1} - AP_{k-1}z\|_2$ instead of

$$\|p - r_{k-1}\|_2, \quad p \in \text{span}\{Ap_1, \ldots, Ap_{k-1}\}^\perp \tag{4.9}$$

is that the optimization problem (4.9) is constrained.
From (4.5) and (4.6), we want to select μ an $\mathbf{w}$ (s) to minimize

$$\left\| \left(1 + \frac{\mu}{\alpha_{k-1}}\right) r_{k-1} + s_{k-1} \right\|^2$$

From (4.8), we have the equivalent

$$\left\| \left(1 + \frac{\mu}{\alpha_{k-1}}\right) r_{k-1} \right\|^2 + \left\| -\frac{\mu}{\alpha_{k-1}}r_{k-2} - AP_{k-2}\mathbf{w} \right\|^2$$

If an optimal solution is $(\mu^*, \mathbf{w}^*)$, then

$$\left\| -\frac{\mu^*}{\alpha_{k-1}} r_{k-2} - AP_{k-2}\mathbf{w}^* \right\|^2 = \left| -\frac{\mu^*}{\alpha_{k-1}} \right|^2 \left\| r_{k-2} - AP_{k-2} \boxed{\frac{\mathbf{w}^*}{-\mu^*/\alpha_{k-1}}} \right\|^2$$

(The boxed term is the optimization problem to find p at $k-1$ steps) and

$$-\mu^*/\alpha_{k-1}$$

must be the optimal solution of

$$\min_z \|r_{k-2} - AP_{k-2}z\|_2$$

From the earlier lemma, the solution of

$$\min_z \|r_{k-2} - AP_{k-2}z\|_2$$

is

$$p_{k-1} = r_{k-2} - AP_{k-2}z_{k-2}$$

Therefore, s_{k-1} is parallel to p_{k-1}, from (4.6), we have

$$p_k \in \text{span}\{r_{k-1}, p_{k-1}\}$$

Now, assume

$$p_k = r_{k-1} + \beta_k p_{k-1} \tag{4.10}$$

This assumption is reasonable because we will adjust the step size α_k to make the direction of p_k correct. Therefore, finding a parallel direction to the real solution of $\min \|p - r_{k-1}\|_2$ is enough.

From the A-conjugacy of p_i and 4.4, we respectively have

$$p_{k-1}^T A p_k = 0, \quad p_{k-1}^T r_{k-1} = 0$$

With (4.10) we have

$$Ap_k = Ar_{k-1} + \beta_k Ap_{k-1}$$
$$0 = p_{k-1}^T Ap_k = p_{k-1}^T Ar_{k-1} + \beta_k p_{k-1}^T Ap_{k-1}$$

So,

$$\beta_k = -\frac{p_{k-1}^T Ar_{k-1}}{p_{k-1}^T Ap_{k-1}}$$

which is a closed-form solution for β_k.

$$\alpha_k = \frac{p_k^T r_{k-1}}{p_k^T Ap_k} = \frac{(r_{k-1} + \beta_k p_{k-1})^T r_{k-1}}{p_k^T Ap_k} = \frac{r_{k-1}^T r_{k-1}}{p_k^T Ap_k}$$

The algorithm is become

CONJUGATE-GRADIENT(A, b)

```
1  k = 0, x_0 = 0, r_0 = b
2  while r_k ≠ 0
3      k = k + 1
4      if k = 1
5          p_1 = r_0
6      else
7          β_k = -(p_{k-1}^T A r_{k-1}) / (p_{k-1}^T A p_{k-1})
8          p_k = r_{k-1} + β_k p_{k-1}
9      α_k = (r_{k-1}^T r_{k-1}) / (p_k^T A p_k)
10     x_k = x_{k-1} + α_k p_k
11     r_k = b - A x_k
```

However, the above algorithm is not efficient because we need to compute three matrix-vector products in each iteration

$$Ap_{k-1}, Ax_k, Ar_{k-1}$$

Lecture 8

4.3.4 Efficient Implementation of Conjugate Gradient Method

2026.05.04

We further simplify the algorithm,

$$\begin{aligned}
r_k &= r_{k-1} - \alpha_k A p_k \\
r_{k-1} &= r_{k-2} - \alpha_{k-1} A p_{k-1} \\
r_{k-1}^T r_{k-1} &= r_{k-1}^T r_{k-2} - \alpha_{k-1} r_{k-1}^T A p_{k-1} \\
&= 0 - \alpha_{k-1} r_{k-1}^T A p_{k-1} \\
r_{k-2}^T r_{k-1} &= r_{k-2}^T r_{k-2} - \alpha_{k-1} r_{k-2}^T A p_{k-1} \\
&= r_{k-2}^T r_{k-2} - \alpha_{k-1} (p_{k-1} - \beta_{k-1} p_{k-2})^T A p_{k-1} \\
&= r_{k-2}^T r_{k-2} - \alpha_{k-1} p_{k-1}^T A p_{k-1} = 0
\end{aligned}$$

Hence, we have

$$r_{k-2}^T r_{k-2} = \alpha_{k-1} p_{k-1}^T A p_{k-1}$$

Now checking β_k, we have

$$\beta_k = \frac{-p_{k-1}^T A r_{k-1}}{p_{k-1}^T A p_{k-1}} = \frac{r_{k-1}^T r_{k-1} / \alpha_{k-1}}{r_{k-2}^T r_{k-2} / \alpha_{k-1}} = \frac{r_{k-1}^T r_{k-1}}{r_{k-2}^T r_{k-2}}$$

The algorithm is become

CONJUGATE-GRADIENT(A, b)

1 $k = 0,\ x_0 = 0,\ r_0 = b$
2 **while** $r_k \neq 0$
3 $\quad k = k + 1$
4 $\quad$ **if** $k = 1$
5 $\quad\quad p_1 = r_0$
6 $\quad$ **else**
7 $\quad\quad \beta_k = -\frac{r_{k-1}^T r_{k-1}}{r_{k-2}^T r_{k-2}}$
8 $\quad\quad p_k = r_{k-1} + \beta_k p_{k-1}$
9 $\quad \alpha_k = \frac{r_{k-1}^T r_{k-1}}{p_k^T A p_k}$
10 $\quad x_k = x_{k-1} + \alpha_k p_k$
11 $\quad r_k = r_{k-1} - \alpha_k A p_k$

Only one matrix-vector product (Ap_k) is needed in each iteration.

> **Note.** We don't need to calculte Ax_k since from the theorem, we have
>
> $$r_k = r_{k-1} - \alpha_k A p_k$$
>
> We can get r_k from Ap_k.

Here, notice that $r_k \neq 0$ is not a practical termination criterion since the numerical error. Also, we should avoid calculating $r_k^T r_k$ twice every iteration since it can be reused from the previous iteration.

Here is the final version of the conjugate gradient method

CONJUGATE-GRADIENT(A, b)

$k = 0,\ x = 0,\ r = b,\ \rho_0 = \|r\|_2^2$
while $\sqrt{\rho_k} > \epsilon\|b\|_2$ and $k < k_{\max}$
    $k = k + 1$
    if $k = 1$
        $p = r$
    else
        $\beta = \frac{\rho_{k-1}}{\rho_{k-2}}$
        $p = r + \beta p$
    $w = Ap$
    $\alpha = \frac{\rho_{k-1}}{p^T w}$
    $x = x + \alpha p$
    $r = r - \alpha w$
    $\rho_k = \|r\|_2^2$

Note. Here change the condition to relative error to avoid the numerical error. But it can also be not achievable since the numerical error can be large. So we also need to set a maximum iteration number. b is a common setting of optimization problems if b is too large, the number of iterations is not n.

Now we analyze the convergence of the conjugate gradient method.

Theorem 4.3.2. If $A = I + B$ is an $n \times n$ symmertic positive definite matrix, and $\text{rank}(B) = r$, then the conjugate gradient method will converge in at most $r + 1$ iterations.

We omit the proof.

Example. $A = I$

$$
\begin{aligned}
&x_0 = 0, r_0 = b \\
&p_1 = r_0 = b \\
&\alpha = r_0^T r_0 / p_1^T A p_1 = b^T b / b^T A b = 1 \\
&x_1 = p_1 = b \\
&r_1 = r_0 - \alpha_1 A p_1 = b - b = 0
\end{aligned}
$$

The conjugate gradient method stops in one iteration.

Definition 4.3.3 (A-norm). We have $\|\cdot\|_A$ defined as

$$\|x\|_A = \sqrt{x^T A x}$$

Theorem 4.3.3. If $Ax = b$,

$$\|x - x_k\|_A \leq 2\|x - x_0\|_A \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k$$

where κ is the condition number of A.

If $\kappa \to 1$, then

$$\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k$$

is smaller.

In general, the convergence of the conjugate gradient method is faster if the condition A is better.

Remark. The property of positive definiteness is used when

$$\min_{x \in \text{span}\{p_1,\ldots,p_k\}} f(x) = \min_y f(x_0 + P_{k-1}y) + \min_\alpha(-\alpha p_k^T r_0 + \frac{\alpha^2}{2} p_k^T A p_k)$$

For the second part,

$$\frac{1}{2}\alpha^2 p^T A p + \cdots$$

we need $p^T A p > 0$ to make the minimization problem well-defined.

Chapter 5

Nonlinear Equations

5.1 One Dimensional Case

Now we consider solving

$$f(x) = 0, \quad x \in \mathbb{R}^1$$

5.1.1 Bisection Method

Here is the procedure

1° Given a_1, b_1 such that

$$f(a_1)f(b_1) < 0$$

Set $i = 1$.

2° At the current iteration $[a_i, b_i]$, compute

$$p_i = \frac{a_i + b_i}{2}$$

If

$$f(p_i) = 0, \quad \text{stop}$$

Else setting

$$\begin{cases} a_{i+1} = a_i, \ b_{i+1} = p_i & \text{if } f(a_i)f(p_i) < 0 \\ a_{i+1} = p_i, \ b_{i+1} = b_i & \text{otherwise} \end{cases}$$

3° Set $i = i + 1$ and go to step 2.

The key is to have

Proposition 5.1.1. If f is continuous, then if

$$f(a_i)f(b_i) < 0$$

then there exists a root in the interval $[a_i, b_i]$.

Here us the code from "Numerical methods" (second edition) by Faires and Burden

```
>> bisect21
This is the Bisection Method.
Input the function F(x) in terms of x
```

```
For example: cos(x)
  'cos(x)'
Input endpoints A < B on separate lines
 0
 1
F(A) and F(B) have same sign
Input endpoints A < B on separate lines
 0
 0.5
F(A) and F(B) have same sign
Input endpoints A < B on separate lines
 0
 2
Input tolerance
 0.001
Input maximum number of iterations - no decimal point
 50
Select output destination
1. Screen
2. Text file
Enter 1 or 2
 1
Select amount of output
1. Answer only
2. All intermediate approximations
Enter 1 or 2
 2
Bisection Method
  I    P                   F(P)
  1    1.00000000e+00     5.4030231e-01
  2    1.50000000e+00     7.0737202e-02
  3    1.75000000e+00    -1.7824606e-01
  4    1.62500000e+00    -5.4177135e-02
  5    1.56250000e+00     8.2962316e-03
  6    1.59375000e+00    -2.2951658e-02
  7    1.57812500e+00    -7.3286076e-03
  8    1.57031250e+00     4.8382678e-04
  9    1.57421875e+00    -3.4224165e-03
 10    1.57226562e+00    -1.4692977e-03
 11    1.57128906e+00    -4.9273569e-04

Approximate solution P =  1.57128906
with F(P) =  -0.00049274
Number of iterations =  11 Tolerance =  1.00000000e-03
```

Our procedure finds a solution close to $\pi/2$.

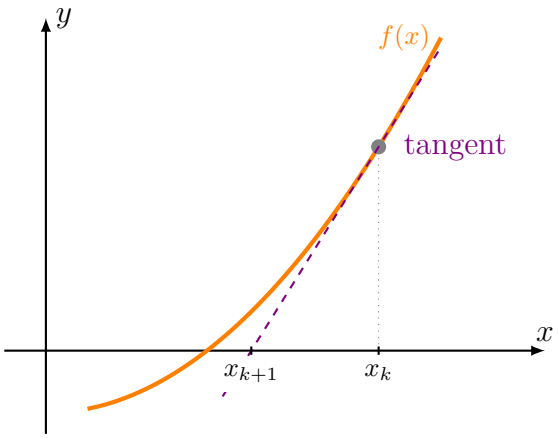


Figure 5.1: The idea of Newton's method

5.1.2 Newton's Method

Here is the rough idea of Newton's method
The idea is to solve

$$f(x) = 0$$

by finding the tangent line at x_k

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

x_k is the current iterate. Then we obtain the next iterate x_{k+1} at the intersection of the tangent line and the x-axis.

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

Newton's method can also be viewed as an optimization method

$$\min_x f(x)$$

This roughly equivalent to solving $f'(x) = 0$

$$\begin{aligned} f(x+d) &= f(x) + f'(x)d + \frac{1}{2}d^2 f''(x) + \cdots \\ &\approx f(x) + f'(x)d + \frac{1}{2}d^2 f''(x) \end{aligned}$$

Instead of minimizing $f(x)$, we can minimize the quadratic approximation

$$\min_d f(x) + f'(x)d + \frac{1}{2}d^2 f''(x)$$

Then

$$\begin{aligned} f''(x)d + f'(x) &= 0 \\ d &= -\frac{f'(x)}{f''(x)} \\ x_{k+1} &= x_k - \frac{f'(x)}{f''(x)}. \end{aligned}$$

However, the convergence of Newton's method is not guaranteed i.e. $\{x_k\}$ may diverge.
Here is an example of divergence of Newton's method.

```
>> newton
newton
This is Newtons Method
Input the function F(x) in terms of x
For example: cos(x)
 'cos(x)'
'cos(x)'
Input the derivative of F(x) in terms of x
 '-sin(x)'
'-sin(x)'
Input initial approximation
 1
1
Input tolerance
 0.001
0.001
Input maximum number of iterations - no decimal point
 50
50
Select output destination
1. Screen
2. Text file
Enter 1 or 2
 1
1
Select amount of output
1. Answer only
2. All intermediate approximations
Enter 1 or 2
 2
2
Newtons Method
  I   P                 F(P)
  1   1.64209262e+00   -7.1235903e-02
  2   1.57067528e+00    1.2104963e-04
  3   1.57079633e+00   -5.9124355e-13

Approximate solution = 1.5707963268e+00
with F(P) = -5.9124355058e-13
Number of iterations = 3
Tolerance = 1.0000000000e-03
```

Another run using a different initial point

```
>> newton
This is Newtons Method
Input the function F(x) in terms of x
For example: cos(x)
 'cos(x)'
```

```
Input the derivative of F(x) in terms of x
 '-sin(x)'
Input initial approximation
 0.1
Input tolerance
 0.001
Input maximum number of iterations - no decimal point
 50
Select output destination
1. Screen
2. Text file
Enter 1 or 2
 1
Select amount of output
1. Answer only
2. All intermediate approximations
Enter 1 or 2
 2
Newtons Method
  I   P                  F(P)
  1   1.00666444e+01   -8.0097972e-01
  2   1.14045284e+01    3.9764987e-01
  3   1.09711401e+01   -2.4431758e-02
  4   1.09955792e+01    4.8638065e-06
  5   1.09955743e+01   -4.2862638e-16

Approximate solution = 1.0995574288e+01
with F(P) = -4.2862637970e-16
Number of iterations = 5
Tolerance = 1.0000000000e-03
```

Our procedure finds a solution close to $\pi/2$, and the iteration number is smaller than Bisection method.

> **Note.** Here we didn't select the step size, which might be a problem since the convergence is not guaranteed.

Lecture 9

5.1.3 Convergence Rate

2026.05.11

Now we want to prove the convergence rate of Newton's method (Quadratic).
First, we assume f satisfies that

1. f is continuously differentiable
2. f' is Lipschitz continuous, i.e.

$$\|f'(y) - f'(x)\| \leq \alpha \|y - x\|, \quad \forall x, y$$

where $\alpha > 0$ is the Lipschitz constant.

Theorem 5.1.1. If $\{x^k\} \to x^*$ and $f'(x^*) \neq 0$, then

1. $f(x^*) = 0$
2. $\exists L \geq 1,\ \delta > 0$ such that $\forall k \geq L$

$$\|x^{k+1} - x^*\| \leq \delta \|x^k - x^*\|^2$$

Note. For this theorem, we call it "local convergence analysis" since we need the sequence $\{x^k\}$ to converge to x^* first.

Proof 1. For the first part, from the update rule of Newton's method and Lemma 5.1.2,

$$\begin{aligned} f(x^{k+1}) &= f(x^k) + f'(x^k)(x^{k+1} - x^k) + e(x^{k+1}, x^k) \\ &= e(x^{k+1}, x^k) \end{aligned}$$

where

$$|e(x^{k+1}, x^k)| \leq \frac{\alpha}{2} \|x^{k+1} - x^k\|^2$$

Thus,

$$|f(x^{k+1})| \leq \frac{\alpha}{2} \|x^{k+1} - x^k\|^2$$

Since $\{x^k\} \to x^*$, and the above inequality holds, we have

$$|f(x^{k+1})| \to 0$$

Since f is continuous, we have

$$f(x^{k+1}) \to f(x^*) \text{ implies } f(x^*) = 0$$

■

Proof 2. For the second part, we define

$$\bar{\beta} \equiv |f'(x^*)^{-1}|$$

Since

$$|x^k - x^*| \to 0$$

and f' is Lipschitz continuous, $\exists L \geq 1$ such that $\forall k \geq L$

$$\begin{aligned} |f'(x^*)^{-1}(f'(x^k) - f'(x^*))| &\leq |f'(x^*)^{-1}||f'(x^k) - f'(x^*)| \\ &\leq \alpha\bar{\beta}|x^k - x^*| \\ &\leq \frac{1}{2} \end{aligned}$$

For $k \geq L$, from Lemma 5.1.1, $f'(x^k)^{-1}$ exists and

$$|f'(x^k)^{-1}| \leq 2\bar{\beta} \tag{5.1}$$

By definition of Newton's method,

$$x^{k+1} - x^* = x^k - x^* - f'(x^k)^{-1} f(x^k) \tag{5.2}$$

Since $f(x^*) = 0$, we have

$$f(x^*) = f(x^k) + f'(x^k)(x^* - x^k) + e(x^*, x^k) = 0$$

Then multiply both sides by $f'(x^k)^{-1}$, we have

$$f'(x^k)^{-1} f(x^k) = x^k - x^* - f'(x^k)^{-1} e(x^*, x^k) \tag{5.3}$$

Thus, (5.1) and (5.2) imply

$$x^{k+1} - x^* = f'(x^k)^{-1} e(x^*, x^k) \tag{5.4}$$

From (5.1) and (5.3), and Lemma 5.1.2, for $k \geq L$,

$$\begin{aligned} |x^{k+1} - x^*| &\leq |f'(x^k)^{-1}||e(x^*, x^k)| \\ &\leq 2\bar{\beta}\frac{1}{2}\alpha|x^k - x^*|^2 \\ &= \delta|x^k - x^*|^2, \end{aligned}$$

where we define

$$\delta = \alpha\bar{\beta}$$

Then proof is complete. ■

Lemma 5.1.1. If

$$|f'(x^*)^{-1}(f'(x^k) - f'(x^*))| \leq 1 \tag{5.5}$$

then

$$|f'(x^k)^{-1}| \leq \frac{|f'(x^*)^{-1}|}{1 - |f'(x^*)^{-1}(f'(x^k) - f'(x^*))|} \tag{5.6}$$

Proof. The inequality (5.6)

$$|f'(x^k)^{-1}| \leq \frac{|f'(x^*)^{-1}|}{1 - |f'(x^*)^{-1}(f'(x^k) - f'(x^*))|}$$

is equivalent to

$$\begin{aligned} & |f'(x_k)^{-1}| - |f'(x^*)^{-1} - f'(x_k)^{-1}| \\ = \; & |f'(x_k)^{-1}| - |f'(x_k)^{-1} - f'(x^*)^{-1}| \\ \leq \; & |f'(x^*)^{-1}|, \end{aligned} \tag{5.7}$$

where the last inequality is from the property of absolute values.

> **Note.** We need the condition (5.5) to ensure that the denominator in (5.6) is positive, which is necessary for the inequality to hold.

Proof complete. ■

Lemma 5.1.2. If f' is Lipschitz continuous with constant $\alpha > 0$, and

$$f(y) = f(x) + f'(x)(y - x) + e(y, x)$$

then

$$|e(y, x)| \leq \frac{\alpha}{2}|y - x|^2$$

Proof. Because

$$\frac{d}{dt}f(x+t(y-x)) = f'(x+t(y-x))(y-x)$$

we have

$$\begin{aligned}
& \int_0^1 (f'(x+t(y-x)) - f'(x))(y-x)dt \\
= & \left(\int_0^1 f'(x+t(y-x))(y-x)dt\right) - f'(x)(y-x) \\
= & f(x+t(y-x))\Big|_{t=0}^{1} - f'(x)(y-x) \\
= & f(y) - f(x) - f'(x)(y-x)
\end{aligned}$$

then

$$\begin{aligned}
|e(y,x)| &\leq \int_0^1 |(f'(x+t(y-x)) - f'(x))(y-x)|dt && (5.8) \\
&\leq \int_0^1 \alpha|(x+t(y-x)) - x||(y-x)|dt && (5.9) \\
&\leq \int_0^1 \alpha t|(y-x)|^2 dt \;=\; \boxed{\frac{1}{2}\alpha|y-x|^2},
\end{aligned}$$

where (5.8) to (5.9) is from the Lipschitz condition of f'. ■

Chapter 6

Interpolation and Approximation

If $f^{(n)}(x)$, $\forall n$ are available, Taylor's polynomial is an approximation:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!}f''(x_0)(x - x_0)^2 + \cdots$$

Example.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

In this approximation, we only need information of one point. If finite terms are used, the approximation can only be accurate near x_0. Primal use of Taylor's polynomial for numerical methods is the derivation of numerical techniques but not for approximation. Subsequently, we discuss some techniques of **interpolation** and **approximation**

Definition 6.0.1 (Interpolation). Find a function passing all the given data points, i.e.

$$f(x_i) = y_i, \quad i = 1, 2, \cdots, n$$

Definition 6.0.2 (Approximation). Find a function that error to the given data points is small, i.e.

$$|f(x_i) - y_i| \leq \epsilon, \quad i = 1, 2, \cdots, n$$

6.1 Interpolation

6.1.1 Lagrange Polynomial

Given $(x_0, f(x_0)), (x_1, f(x_1)), \cdots, (x_n, f(x_n))$, find a polynomial $P_n(x)$ of degree at most n passing through these points.

Note. This is the simplest form of functions.

Degree 1 polynomial passing two points is a straight line.

Example. Given $(x_0, f(x_0)), (x_1, f(x_1))$.

Define

$$L_0(x) \equiv \frac{x - x_1}{x_0 - x_1}, \quad L_1(x) \equiv \frac{x - x_0}{x_1 - x_0}$$

$$P_1(x) = L_0(x)f(x_0) + L_1(x)f(x_1)$$

Then

$$P_1(x_0) = f(x_0), \quad P_1(x_1) = f(x_1)$$

Example. For the general case with $n+1$ points. Consider a polynomial with degree at most n

$$(x_0, f(x_0)), (x_1, f(x_1)), \cdots, (x_n, f(x_n))$$

Definition 6.1.1 (Lagrange polynomial)**.** Construct $L_{n,k}(x)$ as

$$L_{n,k}(x_i) = \begin{cases} 1, & i = k \\ 0, & i \neq k \end{cases}$$

where n is the degree of the polynomial, and k is the index of the point. Then

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

Thus we have

$$P_n(x_i) = f(x_i), \quad i = 0, 1, \cdots, n$$

Definition 6.1.2 (Lagrange basis polynomial)**.** The Lagrange basis polynomial $L_{n,k}(x)$ is defined as

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

6.1.2 Spline Interpolation

Using Lagrange polynomial is bad since

1. The degree of the polynomial is high.
2. There is a large fluctuation between the data points, which is called Runge's phenomenon.

A piecewise polynomial is a better choice.

Definition 6.1.3 (Linear spline)**.** A series of straight lines connecting the data points

$$(x_0, f(x_0)), (x_1, f(x_1)), \cdots, (x_n, f(x_n))$$

Example.

$$x = \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}, y = \begin{bmatrix} 0 & 1 & 4 & 3 \end{bmatrix}$$

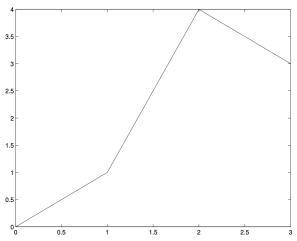


Figure 6.1: Linear spline interpolation

Definition 6.1.4 (Linear spline interpolation). Define $s_0(x), s_1(x), \cdots, s_{n-1}(x)$ as

$$s_j(x) = \frac{x_{j+1} - x}{x_{j+1} - x_j} f(x_j) + \frac{x - x_j}{x_{j+1} - x_j} f(x_{j+1}), \quad x_j \le x \le x_{j+1}$$

However, linear spline is not smooth, which is not differentiable at the data points. Thus, we need a smoother interpolation. A possiblilty is to use higher degree $s_j(x)$ and hope that

$$s_0'(x), s_1'(x), \cdots, s_{n-1}'(x)$$

are continuous.
Thus, the requirement is

$$\begin{aligned} s_j(x_j) = f(x_j), j = 0, \ldots, n-1, \\ s_{n-1}(x_n) = f(x_n) \\ s_{j+1}(x_{j+1}) = s_j(x_{j+1}), j = 0, \ldots, n-2 \\ s_{j+1}'(x_{j+1}) = s_j'(x_{j+1}), j = 0, \ldots, n-2 \end{aligned}$$

Note. We have

$$n + 1 + 2(n-1) = 3n - 1$$

conditions. A quadratic piecewise interpolation has $3n$ parameters, which is more than the number of conditions. For each interval, we can set 3 parameters.

Nowadays, cubic spline interpolation is the most popular choice, which has another name "**Spline**" which use $4n$ parameters.

$$s_j(x_j) = f(x_j), j = 0, \ldots, n-1, \tag{6.1}$$
$$s_{n-1}(x_n) = f(x_n)$$
$$s_{j+1}(x_{j+1}) = s_j(x_{j+1}), j = 0, \ldots, n-2 \tag{6.2}$$
$$s_{j+1}'(x_{j+1}) = s_j'(x_{j+1}), j = 0, \ldots, n-2$$
$$s_{j+1}''(x_{j+1}) = s_j''(x_{j+1}), j = 0, \ldots, n-2$$

The number of conditions is

$$n + 1 + 2(n-1) + (n-2) = 4n - 2$$

From the view point of solving the system of equations, we have $4n$ parameters and $4n-2$ conditions, which means we need to set 2 more conditions to make the system solvable.
The boundary conditions are

$$\begin{aligned} s_0''(x_0) = 0, s_{n-1}''(x_n) = 0 \quad \text{(natural spline)} \\ s_0'(x_0) = f'(x_0), s_{n-1}'(x_n) = f'(x_n) \quad \text{(clamped spline)} \end{aligned}$$

Now we have $4n$ conditions, hence the system is solvable.

Definition 6.1.5 (Cubic spline interpolation). Construct cubic polynomials

$$s_j(x) \equiv a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3, \quad j = 0, 1, \cdots, n-1$$

Immediately,

$$s_j(x_j) = a_j = f(x_j), \quad j = 0, 1, \cdots, n-1$$

Now (6.2) becomes

$$a_{j+1} = a_j + b_j(x_{j+1} - x_j) + c_j(x_{j+1} - x_j)^2 + d_j(x_{j+1} - x_j)^3, \quad j = 0, 1, \cdots, n-2$$

Now we define

$$h_j \equiv x_{j+1} - x_j$$
$$a_n \equiv f(x_n)$$

Earlier a_j is only from 0 to $n-1$.
We have

$$a_n = s_{n-1}(x_n) = a_{n-1} + b_{n-1}h_{n-1} + c_{n-1}h_{n-1}^2 + d_{n-1}h_{n-1}^3$$

Then

$$a_{j+1} = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3, \quad j = 0, 1, \cdots, n-1 \tag{6.3}$$

Now we define

$$b_n \equiv s'_{n-1}(x_n) \tag{6.4}$$

Earlier b_j is only from 0 to $n-1$.
Using

$$\begin{aligned} s'_j(x) &= b_j + 2c_j(x - x_j) + 3d_j(x - x_j)^2 \\ s'_{j+1}(x_{j+1}) &= s'_j(x_{j+1}), \quad j = 0, \ldots, n-2 \\ b_{j+1} &= b_j + 2c_j h_j + 3d_j h_j^2, \quad j = 0, \ldots, n-1 \end{aligned} \tag{6.5}$$

Then we define

$$c_n \equiv s''_{n-1}(x_n)/2$$

Since

$$\begin{aligned} s''_{j+1}(x_{j+1}) &= s''_j(x_{j+1}), \quad j = 0, \ldots, n-2 \\ s''_j(x) &= 2c_j + 6d_j(x - x_j) \\ s''_j(x_{j+1}) &= 2c_j + 6d_j h_j \\ s''_{j+1}(x_{j+1}) &= 2c_{j+1} \end{aligned}$$

we have

$$c_{j+1} = c_j + 3d_j h_j, \quad j = 0, \ldots, n-1$$

Thus

$$d_j = \frac{c_{j+1} - c_j}{3h_j}$$

Note. This process is like Gaussian elimination, where we eliminate a_j and b_j to get a system of equations with only c_j.

Hence, (6.3) becomes

$$\begin{aligned} a_{j+1} &= a_j + b_j h_j + c_j h_j^2 + \frac{c_{j+1} - c_j}{3h_j} h_j^3 \\ &= a_j + b_j h_j + \frac{h_j^2}{3}(2c_j + c_{j+1}), \quad j = 0, \ldots, n-1 \end{aligned} \tag{6.6}$$

And (6.5) becomes

$$\begin{aligned} b_{j+1} &= b_j + 2c_j h_j + 3d_j h_j^2 \\ &= b_j + 2c_j h_j + 3\frac{c_{j+1}-c_j}{3h_j} h_j^2 \\ &= b_j + 2c_j h_j + h_j(c_{j+1}-c_j) \\ &= b_j + h_j(c_j + c_{j+1}), \quad j = 0, \ldots, n-1 \end{aligned} \tag{6.7}$$

We have known a_j, unknown b_j and c_j.
From (6.6)

$$b_j = \frac{1}{h_j}(a_{j+1} - a_j) - \frac{h_j}{3}(2c_j + c_{j+1}), \quad j = 0, 1, \cdots, n-1$$

i.e.

$$b_{j-1} = \frac{1}{h_{j-1}}(a_j - a_{j-1}) - \frac{h_{j-1}}{3}(2c_{j-1} + c_j), \quad j = 1, 2, \cdots, n$$

Rewriting (6.7)

$$b_j = b_{j-1} + h_{j-1}(c_{j-1} + c_j), \quad j = 1, 2, \cdots, n$$

and substituting the expression

$$\frac{1}{h_j}(a_{j+1} - a_j) - \frac{h_j}{3}(2c_j + c_{j+1}) = \frac{1}{h_{j-1}}(a_j - a_{j-1}) - \frac{h_{j-1}}{3}(2c_{j-1} + c_j) + h_{j-1}(c_{j-1} + c_j), j = 1, \ldots, n-1$$

Note. j cannot be zero now.

Finally, we have

$$\frac{h_{j-1}}{3}c_{j-1} + \frac{2(h_{j-1}+h_j)}{3}c_j + \frac{h_j}{3}c_{j+1} = \frac{1}{h_j}(a_{j+1} - a_j) - \frac{1}{h_{j-1}}(a_j - a_{j-1})$$

so,

$$\begin{aligned} & h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} \\ = \; & \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}), j = 1, \ldots, n-1 \end{aligned} \tag{6.8}$$

The only unknowns are c_j, $j = 0, 1, \cdots, n$. Once we solve c_j, we can solve b_j and d_j. We see the boundary conditions. If $s_0''(x_0) = s_{n-1}''(x_n) = 0$, then

$$s_0''(x) = 2c_0 + 6d_0(x - x_0)$$

$$s_0''(x_0) = 2c_0 = 0 \Rightarrow c_0 = 0, c_n = 0$$

Solving c_j, $j = 0, 1, \cdots, n$ using $(n+1)$ equalities: 6.8 and the above boundary conditions. (with Thomas algorithm for banded linear system)

Remark. We can use other boundary conditions, such as

$$s_0'(x_0) = f'(x_0) \text{ and } s_{n-1}'(x_n) = f'(x_n)$$

Example. Example in MATLAB

```
>> x = [-2 -1 0 1 2]';
>> y = [4 -1 2 1 8]';
>> yy = interp1(x,y,-2:0.1:2,'spline') ;
>> plot(-2:0.1:2,yy);
```

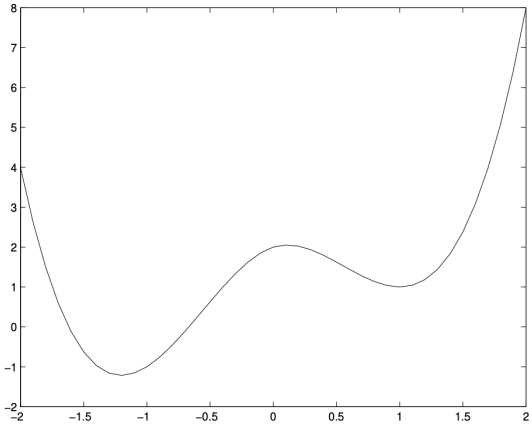


Figure 6.2: Spline interpolation

```
>> [sp] = spline(x,y)
sp =
      form: 'pp'
    breaks: [-2 -1 0 1 2]
     coefs: [4x4 double]
    pieces: 4
     order: 4
       dim: 1
```

- `spline`: returns the spline structure
- `interp1`: 1-D interpolation and returns the approximated values at the query points.

6.2 Approximation

Lecture 10

6.3 Approximation

2026.05.18

6.3.1 Function Approximation

Given data points $\{(x_i, y_i)\}_{i=1}^n$, we want to find a function f such that

$$y_i = f(x_i), \quad i = 1, 2, \ldots, n.$$

However, the data may contain noise, so there is no need to find a function that fits the data points exactly. Instead, we can find a function that approximates the data points well.
The problem becomes minimizing the error

$$E(f) = \min_f \sum_{i=1}^n (y_i - f(x_i))^2 \tag{6.9}$$

or

$$E(f) = \min_f \sum_{i=1}^n |y_i - f(x_i)|$$

with

$$f(x) = ax + b$$

Remark. We have to make some restrictions on the function f to avoid arbitrary solutions.

Assume that $x \in \mathbb{R}^1, y \in \mathbb{R}^1$.

Note. Square error is more commonly used than absolute error because it is easier to optimize with its differentiability.

Then 7.7 becomes "least squares" problem.

6.3.2 Least Squares Approximation

Definition 6.3.1 (Least Squares). Now the goal becomes to find a and b such that

$$\min_{a,b} E = \min_{a,b} \sum_{i=1}^{n} (y_i - (ax_i + b))^2$$

Then we rewrite the objective to let constants in the first term

$$\begin{aligned}
\min_{a,b} \sum_{i=1}^{n} (y_i - (ax_i + b))^2 &\equiv \min_{a,b} \sum_{i=1}^{n} y_i^2 - 2y_i(ax_i + b) + (ax_i + b)^2 \\
&\equiv \min_{a,b} \sum_{i=1}^{n} -2y_i(ax_i + b) + a^2x_i^2 + 2abx_i + b^2 \\
&\equiv \min_{a,b} \left((\sum_{i=1}^{n} -2y_ix_i)a + (\sum_{i=1}^{n} -2y_i)b + (\sum_{i=1}^{n} x_i^2)a^2 + (\sum_{i=1}^{n} 2x_i)ab + nb^2 \right)
\end{aligned}$$

We do the partial derivatives with respect to a and b and set them to zero to find the optimal solution. We have

$$\frac{\partial E}{\partial a} = \sum_{i=1}^{n} -2y_ix_i + 2\left(\sum_{i=1}^{n} x_i^2\right) a + \left(\sum_{i=1}^{n} 2x_i\right) b = 0$$

and

$$\frac{\partial E}{\partial b} = \sum_{i=1}^{n} -2y_i + \left(\sum_{i=1}^{n} 2x_i\right) a + 2nb = 0$$

We have two equations and two unknowns, so we can solve for a and b. Rearranging the equations, we define

Definition 6.3.2.

$$S_{xx} = \sum_{i=1}^{n} x_i^2, \quad S_x = \sum_{i=1}^{n} x_i$$
$$S_{xy} = \sum_{i=1}^{n} x_iy_i, \quad S_y = \sum_{i=1}^{n} y_i$$

The equations become

$$S_{xx}a + S_xb = S_{xy}$$
$$S_xa + nb = S_y$$

Then solve the problem by Gaussian elimination:

$$a = \frac{nS_{xy} - S_xS_y}{nS_{xx} - S_x^2}$$
$$b = \frac{S_{xx}S_y - S_xS_{xy}}{nS_{xx} - S_x^2}$$

Now the problem becomes finding that "Does $\nabla E = 0$ guarantee that we have found the minimum?" For this problem, we need secondary derivatives (i.e., $\nabla^2 E$) to be positive semi-definite.

$$\nabla^2 E = \begin{pmatrix} \frac{\partial^2 E}{\partial a^2} & \frac{\partial^2 E}{\partial a \partial b} \\ \frac{\partial^2 E}{\partial b \partial a} & \frac{\partial^2 E}{\partial b^2} \end{pmatrix} = \begin{pmatrix} 2S_{xx} & 2S_x \\ 2S_x & 2n \end{pmatrix}$$

> **Note.** Why semi-definite can guarantee that we have found the minimum? Because if $\nabla^2 E$ is positive semi-definite, then E is convex, and any critical point of a convex function is a global minimum.

We have $S_{xx} \geq 0$ and $n > 0$, and determinant

$$\det\left(\nabla^2 E\right) = S_{xx} n - S_x^2 = \sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} 1 - \left(\sum_{i=1}^{n} x_i\right)^2 \geq 0$$

Remember Cauchy-Schwarz inequality:

$$\left(\sum_{i=1}^{n} a_i b_i\right)^2 \leq \left(\sum_{i=1}^{n} a_i^2\right)\left(\sum_{i=1}^{n} b_i^2\right)$$

Now consdier an extension "Nonlinear least squares"

Definition 6.3.3 (Nonlinear least squares)**.** Define

$$E(a, b) = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

where $f(x) = ax^2 + bx + c$. Then we solve

$$\min_f E = \min_{a,b,c} E(a, b, c)$$

The situation is similar to linear least squares.

6.3.3 Higher Dimensional Space

Definition 6.3.4. If $x \in \mathbb{R}^m,\ m > 1$ then

$$f(x) = a^\top x + b,\ a \in \mathbb{R}^m$$

Let

$$E = \sum_{i=1}^{n} (y_i - a^\top x_i - b)^2$$

Then the problem is

$$\begin{aligned} \min_{a,b} \sum_{i=1}^{n} (y_i - (a^\top x_i + b))^2 &\equiv \min_{a,b} \sum_{i=1}^{n} y_i^2 - 2y_i(a^T x_i + b) + (a^\top x_i + b)^2 \\ &\equiv \min_{a,b} \sum_{i=1}^{n} -2y_i(a^\top x_i + b) + (a^\top x_i)^2 + 2ba^\top x_i + b^2 \\ &\equiv \min_{a,b} (\sum_{i=1}^{n} -2y_i x_i)^\top a + (\sum_{i=1}^{n} -2y_i) b + \sum_{i=1}^{n} (a^\top x_i)^2 + (\sum_{i=1}^{n} 2x_i)^\top ab + nb^2 \end{aligned}$$

Then we do the partial derivatives with respect to a and b and set them to zero to find the optimal solution. We have

$$\frac{\partial E}{\partial a} = \sum_{i=1}^{n} -2y_i x_i + 2\sum_{i=1}^{n}(a^\top x_i)x_i + \left(\sum_{i=1}^{n} 2x_i\right) b = 0$$

and

$$\frac{\partial E}{\partial b} = \sum_{i=1}^{n} -2y_i + \left(\sum_{i=1}^{n} 2x_i\right)^\top a + 2nb = 0$$

Note.

$$\sum_{i=1}^{n}(a^\top x_i)x_i = \sum_{i=1}^{n} x_i x_i^\top a$$

where

$$x_i x_i^\top \in \mathbb{R}^{m\times m}$$

Definition 6.3.5. Define

$$S_{xx} = \sum_{i=1}^{n} x_i x_i^\top, S_x = \sum_{i=1}^{n} x_i$$
$$S_{xy} = \sum_{i=1}^{n} y_i x_i, S_y = \sum_{i=1}^{n} y_i$$

The equations become solving linear system of a and b:

$$\begin{pmatrix} S_{xx} & S_x \\ S_x^\top & n \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} S_{xy} \\ S_y \end{pmatrix}$$

which contains $m+1$ variables.

Recall. We solve

$$\min_{a,b}(y_i - (a^\top x_i + b))^2$$

We have

$$a^\top x_i = \begin{pmatrix} a^\top & b \end{pmatrix}\begin{pmatrix} x_i \\ 1 \end{pmatrix}$$

We can consider that each input vector has one extra dimension with value 1.
Hence, we can define

$$\bar{a} = \begin{pmatrix} a \\ b \end{pmatrix}, \quad \bar{x}_i = \begin{pmatrix} x_i \\ 1 \end{pmatrix}$$

The linear system becomes

$$S_{\bar{x}\bar{x}}\bar{a} = S_{\bar{x}y}$$

We have

$$S_{\bar{x}\bar{x}} = \sum_{i=1}^{n} \bar{x}_i \bar{x}_i^\top = \sum_{i=1}^{n}\begin{pmatrix} x_i \\ 1 \end{pmatrix}\begin{pmatrix} x_i^\top & 1 \end{pmatrix} = \begin{pmatrix} S_{xx} & S_x \\ S_x^\top & n \end{pmatrix}$$

which is the matrix we obtain in the previous linear system. Because each

$$\bar{x}_i \bar{x}_i^\top$$

is positive semi-definite, we easily know that $S_{\bar{x}\bar{x}}$ is positive semi-definite.

Chapter 7

Fast Fourier Transformation

7.1 Basic Methods

7.1.1 Continuous Least Squares

Definition 7.1.1 (Continuous least squares). Given $f \in C[a,b]$, i.e., continuous function on $[a,b]$. Find the polinomial

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0 = \sum_{k=0}^{n} a_k x^k$$

to minimize

$$E = \int_a^b (f(x) - p_n(x))^2 \, \mathrm{d}x = \int_a^b \left(f(x) - \sum_{k=0}^{n} a_k x^k \right)^2 \, \mathrm{d}x$$

Let

$$\frac{\partial E}{\partial a_j} = 0, \quad j = 0, 1, \ldots, n$$

Now

$$E = \int_a^b f(x)^2 \, \mathrm{d}x - 2\sum_{k=0}^{n} \int_a^b a_k x^k f(x) \, \mathrm{d}x + \int_a^b \left(\sum_{k=0}^{n} a_k x^k \right)^2 \, \mathrm{d}x$$

So,

$$\begin{aligned} \frac{\partial E}{\partial a_j} &= -2 \int_a^b x^j f(x) \, \mathrm{d}x + \int_a^b 2 \left(\sum_{k=0}^{n} a_k x^k \right) x^j \, \mathrm{d}x \\ &= -2 \int_a^b x^j f(x) \, \mathrm{d}x + 2 \sum_{k=0}^{n} a_k \int_a^b x^{j+k} \, \mathrm{d}x = 0 \end{aligned}$$

Then we obtain the linear system with $n+1$ linear equations

$$\sum_{k=0}^{n} a_k \int_a^b x^{j+k} \, \mathrm{d}x = \int_a^b x^j f(x) \, \mathrm{d}x, \quad j = 0, 1, \ldots, n$$

Example. Let $f(x) = \sin(\pi x)$ on $[0, 1]$.

We have approximation by degree 2 polynomial

$$a_0 \int_0^1 x^0 \, \mathrm{d}x + a_1 \int_0^1 x^1 \, \mathrm{d}x + a_2 \int_0^1 x^2 \, \mathrm{d}x = \int_0^1 x^0 \sin(\pi x) \, \mathrm{d}x$$
$$a_0 \int_0^1 x^1 \, \mathrm{d}x + a_1 \int_0^1 x^2 \, \mathrm{d}x + a_2 \int_0^1 x^3 \, \mathrm{d}x = \int_0^1 x^1 \sin(\pi x) \, \mathrm{d}x$$
$$a_0 \int_0^1 x^2 \, \mathrm{d}x + a_1 \int_0^1 x^3 \, \mathrm{d}x + a_2 \int_0^1 x^4 \, \mathrm{d}x = \int_0^1 x^2 \sin(\pi x) \, \mathrm{d}x$$

Note.

$$\begin{aligned} \int_0^1 \sin \pi x \, \mathrm{d}x &= \frac{-1}{\pi} \cos \pi x \Big|_0^1 \\ &= \frac{-1}{\pi}(-1 - 1) = \frac{2}{\pi} \end{aligned}$$

$$\begin{aligned} \int_0^1 x \sin \pi x \, \mathrm{d}x &= \frac{-1}{\pi} x \cos \pi x \Big|_0^1 - \int_0^1 \frac{-1}{\pi} \cos \pi x \, \mathrm{d}x \\ &= \frac{1}{\pi} + \frac{1}{\pi}\frac{1}{\pi} \sin \pi x \Big|_0^1 \\ &= \frac{1}{\pi} \end{aligned}$$

$$\begin{aligned} \int_0^1 x^2 \sin \pi x \, \mathrm{d}x &= \frac{-1}{\pi} x^2 \cos \pi x \Big|_0^1 - \frac{-1}{\pi} \int_0^1 2x \cos \pi x \, \mathrm{d}x \\ &= \frac{1}{\pi} + \frac{2}{\pi}\left(\frac{1}{\pi} x \sin \pi x \Big|_0^1 - \int_0^1 \frac{1}{\pi} \sin \pi x \, \mathrm{d}x\right) \\ &= \frac{1}{\pi} + \frac{2}{\pi}\left(0 + \frac{1}{\pi^2} \cos \pi x \Big|_0^1\right) \\ &= \frac{1}{\pi} + \frac{2}{\pi}\left(\frac{-2}{\pi^2}\right) = \frac{1}{\pi} - \frac{4}{\pi^3} \end{aligned}$$

Solution of three equations is

$$a_0 = \frac{12\pi^2 - 120}{\pi^3} \approx -0.050465$$
$$a_1 = -a_2 = \frac{720 - 60\pi^2}{\pi^3} \approx 4.12251$$

The approximation is

$$p_2(x) = -4.12251x^2 + 4.12251x - 0.050465$$

We can write a program to check the approximation.

```
x = 0:0.1:1;
y = sin(pi*x);
p = [-4.12251 4.12251  -0.050465];
z = polyval(p,x);
plot(x,y,'r-', x, z, 'b--')
```

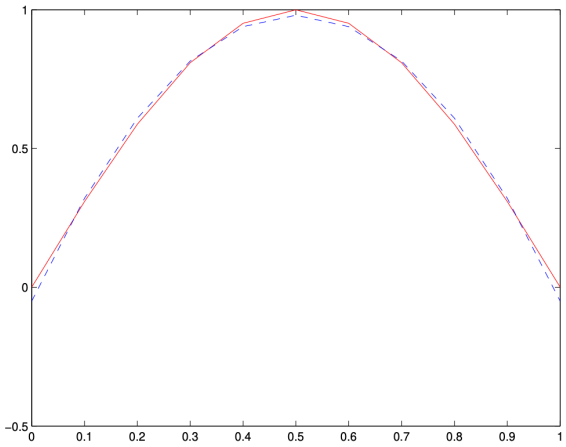


Figure 7.1: Approximation of $\sin(\pi x)$ by a degree 2 polynomial

However, solving a $(n+1) \times (n+1)$ linear system is very time-consuming.

7.1.2 Linearly Independent Functions

Definition 7.1.2 (Linearly independent functions)**.** Functions $\{\phi_0, \phi_1, \cdots, \phi_n\}$ are linearly independent on $[a, b]$ if, whenever

$$c_0\phi_0(x) + c_1\phi_1(x) + \cdots + c_n\phi_n(x) = 0, \quad \forall x \in [a, b]$$

we have $c_0 = c_1 = \cdots = c_n = 0$.

Theorem 7.1.1. For linearly independent set of polynomials. If

$$\phi_j \text{ is a polynomial of degree } j,$$

then

$$\{\phi_0, \phi_1, \cdots, \phi_n\} \text{ are linearly independent on } [a, b].$$

Proposition 7.1.1. Any polynomial of degree $\leq n$, can be written uniquely as a linear combination of $\{\phi_0, \phi_1, \cdots, \phi_n\}$.

If $\{\phi_0, \phi_1, \cdots, \phi_n\}$ are linearly independent, and

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

then

$$\min_{a_0, a_1, \cdots, a_n} E = \int_a^b \left(f(x) - \sum_{k=0}^{n} a_k\phi_k(x) \right)^2 \mathrm{d}x$$

lead to

$$0 = -\frac{\partial E}{\partial a_j} = 2\int_a^b \left(f(x) - \sum_{k=0}^{n} a_k\phi_k(x) \right) \phi_j(x) \, \mathrm{d}x$$

For $j = 0, 1, \ldots, n$, we have

$$\sum_{k=0}^{n} a_k \int_a^b \phi_k(x)\phi_j(x) \, \mathrm{d}x = \int_a^b f(x)\phi_j(x) \, \mathrm{d}x \tag{7.1}$$

Definition 7.1.3. If $\{\phi_0, \phi_1, \cdots, \phi_n\}$ are chosen so that

$$\int_a^b \phi_k(x)\phi_j(x)\ \mathrm{d}x = \begin{cases} 0, & \text{if } j \neq k \\ \alpha_k > 0, & \text{if } j = k \end{cases}$$

we say they are orthogonal on $[a, b]$.

If $\alpha_k = 1$ for $k = 0, 1, \ldots, n$, we say they are orthonormal on $[a, b]$.

Note. The reason α_k has to be positive is

$$\int_a^b \phi_k(x)^2\ \mathrm{d}x > 0$$

The eqs. 7.1 becomes

$$a_j\alpha_j = \int_a^b f(x)\phi_j(x)\ \mathrm{d}x$$

i.e.,

$$a_j = \frac{1}{\alpha_j}\int_a^b f(x)\phi_j(x)\ \mathrm{d}x$$

which is a simple formula with only one variable a_j to solve.

7.1.3 Trigonometric Polynomials Approximation

Theorem 7.1.2. Consider $\{\phi_0, \phi_1, \cdots, \phi_{2n-1}\}$ such that

$$\phi_0(x) = \frac{1}{2}$$

$$\phi_k(x) = \cos kx, \quad k = 1, 2, \ldots, n$$

$$\phi_{n+k}(x) = \sin kx, \quad k = 1, 2, \ldots, n-1$$

is orthogonal on $[-\pi, \pi]$.

Proof. If $k \neq j$, $k \in \{1, 2, \ldots, n\}$, $j \in \{1, 2, \ldots, n\}$, we have

$$\int_{-\pi}^{\pi} \phi_{n+k}(x)\phi_j(x)\ \mathrm{d}x = \int_{-\pi}^{\pi} \sin kx \cos jx\ \mathrm{d}x$$

Using

$$\sin kx \cos jx = \frac{1}{2}(\sin(k+j)x + \sin(k-j)x)$$

Plug in

$$\begin{aligned}\int_{-\pi}^{\pi} \phi_{n+k}(x)\phi_j(x)\ \mathrm{d}x &= \frac{1}{2}\int_{-\pi}^{\pi} (\sin(k+j)x + \sin(k-j)x)\ \mathrm{d}x \\ &= \frac{1}{2}\left(\frac{-\cos(k+j)x}{k+j} + \frac{-\cos(k-j)x}{k-j}\right)\Bigg|_{-\pi}^{\pi} = 0\end{aligned}$$

If $k = j$,

$$\begin{aligned}\int_{-\pi}^{\pi} \sin kx \cos kx\ \mathrm{d}x &= \frac{1}{2}\int_{-\pi}^{\pi} (\sin 2kx)\ \mathrm{d}x \\ &= \frac{-1}{4k}\cos 2kx\Big|_{-\pi}^{\pi} = 0\end{aligned}$$

There are two other cases:

- $k, j,\ k = 1, 2, \ldots, n,\ j = 1, 2, \ldots, n$
- $n + k, n + j,\ k = 1, 2, \ldots, n - 1,\ j = 1, 2, \ldots, n - 1$

Note. Some derivations include $\sin nx$ as well, though it is not necessary for the approximation.

Also we need to check $k = 0$. But they are all zero. Hence, we have shown that $\{\phi_0, \phi_1, \cdots, \phi_{2n-1}\}$ are orthogonal on $[-\pi, \pi]$. ■

Let

$$S_n(x) = \frac{a_0}{2} + a_n \cos nx + \sum_{k=1}^{n-1} (a_k \cos kx + b_k \sin kx)$$

With the orthalogonality, we have

$$a_k = \frac{\int_{-\pi}^{\pi} f(x) \cos kx \ \mathrm{d}x}{\int_{-\pi}^{\pi} \cos^2 kx \ \mathrm{d}x} = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos kx \ \mathrm{d}x, \quad k = 0, 1, \ldots, n \tag{7.2}$$

$$b_k = \frac{\int_{-\pi}^{\pi} f(x) \sin kx \ \mathrm{d}x}{\int_{-\pi}^{\pi} \sin^2 kx \ \mathrm{d}x} = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin kx \ \mathrm{d}x, \quad k = 1, 2, \ldots, n - 1$$

For the denominators, we can easily check that

$$\int_{-\pi}^{\pi} \cos^2 kx \ \mathrm{d}x = \int_{-\pi}^{\pi} \sin^2 kx \ \mathrm{d}x = \pi$$

Note. The reason we use $\frac{a_0}{2}$ is that

$$a_0 = \frac{\int_{-\pi}^{\pi} f(x) \frac{1}{2} \ \mathrm{d}x}{\int_{-\pi}^{\pi} \left(\frac{1}{2}\right)^2 \ \mathrm{d}x} = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \ \mathrm{d}x = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos 0 \cdot x \ \mathrm{d}x$$

Then, it becomes consistent with eqs. 7.2.

Remark. Why considering $\cos kx$ as well as $\sin kx$? But not $\cos kx$ alone? Because for the convergence theorem of Fourier series, we need to consider both $\cos kx$ and $\sin kx$.

7.1.4 Fourier Series

The function we consider is

$$S_n(x) = \frac{a_0}{2} + a_n \cos nx + \sum_{k=1}^{n-1} (a_k \cos kx + b_k \sin kx)$$

When $n \to \infty$, we have

$$\lim_{n \to \infty} S_n(x)$$

is called the "Fourier series" of $f(x)$. Under some conditions, we can show that

$$S_n(x) \to f(x), \quad \text{as } n \to \infty$$

Consider

$$f(x) = |x| \quad \forall x \in (-\pi, \pi)$$

$$a_0 = \frac{1}{\pi}\int_{-\pi}^{\pi} |x| \ \mathrm{d}x = \frac{2}{\pi}\int_0^{\pi} x \ \mathrm{d}x = \pi$$
$$a_k = \frac{1}{\pi}\int_{-\pi}^{\pi} |x|\cos kx \ \mathrm{d}x = \frac{2}{\pi}\int_0^{\pi} x\cos kx \ \mathrm{d}x \tag{7.3}$$

Then

$$\begin{aligned}\int_0^{\pi} x\cos kx \ \mathrm{d}x &= x\frac{\sin kx}{k}\bigg|_0^{\pi} - \int_0^{\pi}\frac{\sin kx}{k} \ \mathrm{d}x \\ &= \frac{\pi\sin k\pi}{k} + \frac{\cos kx}{k^2}\bigg|_0^{\pi} = \frac{(-1)^k - 1}{k^2}\end{aligned}$$

Therefore, from eq. 7.3, we have

$$a_k = \frac{2}{\pi k^2}((-1)^k - 1), \quad \forall k = 1, 2, \dots$$

Because

$$\sin x = -\sin(-x)$$

we have

$$|x|\sin x = -|-x|\sin(-x)$$

and therefore

$$b_k = \frac{1}{\pi}\int_{-\pi}^{\pi} |x|\sin kx \ \mathrm{d}x = 0, \quad \forall k = 1, 2, \dots$$

Finally,

$$S_n(x) = \frac{\pi}{2} + \frac{2}{\pi}\sum_{k=1}^{n}\frac{(-1)^k - 1}{k^2}\cos kx$$

We can see

$$\begin{aligned}S_0(x) &= \frac{\pi}{2}, \text{ a straight line} \\ S_1(x) &= \frac{\pi}{2} - \frac{4}{\pi}\cos x \\ S_2(x) &= \frac{\pi}{2} - \frac{4}{\pi}\cos x + 0 \\ S_3(x) &= \frac{\pi}{2} - \frac{4}{\pi}\cos x - \frac{4}{9\pi}\cos 3x\end{aligned}$$

> **Note.** This example we only use $\cos kx$ because $b_k = 0$ for all k. In general, we need to use both $\cos kx$ and $\sin kx$ to approximate a function.

Lecture 11

7.1.5 Discrete Analog of Fourier Series

2026.05.25

Calculating the integration is very difficult in numerical coputing. Thus, we just consider some points on the function.

Given

$$\{x_j, y_j\}_{j=0}^{2m-1}$$

Consider $-\pi \le x \le \pi$

$$x_j = -\pi + \left(\frac{j}{m}\right)\pi, \quad j = 0, 1, \dots, 2m-1$$

Assume $n < m$. Consider

$$\{\phi_0, \dots, \phi_{2n-1}\}$$

such that

$$\phi_0(x) = \frac{1}{2}$$
$$\phi_k(x) = \cos kx, k = 1, \ldots, n$$
$$\phi_{n+k}(x) = \sin kx, k = 1, \ldots, n-1$$

Definition 7.1.4 (Discrete Fourier Series). The apporximation function $S_n(x)$

$$\begin{aligned} S_n(x) &= \frac{a_0}{2} + \sum_{k=1}^{n} a_k \cos kx + \sum_{k=1}^{n-1} b_k \sin kx \\ &= \frac{a_0}{2} + a_n \cos nx + \sum_{k=1}^{n-1} (a_k \cos kx + b_k \sin kx) \end{aligned}$$

The problem of least square approximation is to find $a_0, a_1, \ldots, a_n$ and $b_1, \ldots, b_{n-1}$ to minimize

$$E = \sum_{j=0}^{2m-1} (y_j - S_n(x_j))^2$$

That is

$$\min_{a_0, a_1, \ldots, a_n, b_1, \ldots, b_{n-1}} \sum_{j=0}^{2m-1} \left(y_j - \frac{a_0}{2} - a_n \cos nx_j - \sum_{r=1}^{n-1} (a_r \cos rx_j + b_r \sin rx_j) \right)^2$$

The situation becomes a multivariable optimization problem. First consider $1 \le k \le n$:

$$\begin{aligned} -\frac{\partial E}{\partial a_k} &= \sum_{j=0}^{2m-1} 2\left(y_j - (\frac{a_0}{2} + a_n \cos nx_j + \sum_{r=1}^{n-1} (a_r \cos rx_j + b_r \sin rx_j)) \right) \cos kx_j \\ &= 2 \sum_{j=0}^{2m-1} y_j \cos kx_j - 2a_k \sum_{j=0}^{2m-1} \cos kx_j \cos kx_j - 2 \sum_{r=1, r\neq k}^{n} \sum_{j=0}^{2m-1} a_r \cos rx_j \cos kx_j - \\ &\quad \sum_{j=0}^{2m-1} a_0 \cos kx_j - 2 \sum_{r=1}^{n-1} \sum_{j=0}^{2m-1} b_r \sin rx_j \cos kx_j \end{aligned}$$

If

$$\sum_{j=0}^{2m-1} \cos rx_j \cos kx_j = 0, \ k \neq r \tag{7.4}$$

$$\sum_{j=0}^{2m-1} \sin rx_j \sin kx_j = 0, \ 1 \le r \le n-1, \ 1 \le k \le n-1 \tag{7.5}$$

and

$$\sum_{j=0}^{2m-1} \cos kx_j = 0, \tag{7.6}$$

then we have

$$\frac{\partial E}{\partial a_k} = 2 \sum_{j=0}^{2m-1} y_j \cos kx_j - 2a_k \sum_{j=0}^{2m-1} \cos^2 kx_j = 0$$

Similar to the continuous case, we hope

$$\phi_0, \ldots, \phi_{2n-1}$$

are orthogonal with respect to the summation over equally spaced points $x_0, \ldots, x_{2m-1}$ in $[-\pi, \pi]$.

To prove (7.4) to (7.6), we need the following theorem.

Theorem 7.1.3. If r isn't a multiple of $2m$, then

$$\sum_{j=0}^{2m-1} \cos rx_j = 0 \text{ and } \sum_{j=0}^{2m-1} \sin rx_j = 0$$

Here is the proof of (7.4). From

$$\cos rx_j \cos kx_j = \frac{1}{2}(\cos(r+k)x_j + \cos(r-k)x_j)$$

Now

$$1 \le r \le n,\ 1 \le k \le n,\ r \ne k$$

then

$$1 \le r + k \le 2n < 2m$$

so $r + k$ isn't a multiple of $2m$. From the theorem, we have

$$\sum_{j=0}^{2m-1} \cos(r+k)x_j = 0$$

For $r - k$, we have

$$-2m < -n \le r - k \ne 0 \le n < 2m$$

so $r - k$ isn't a multiple of $2m$. Thus,

$$\sum_{j=0}^{2m-1} \cos(r-k)x_j = 0$$

Therefore,

$$\sum_{j=0}^{2m-1} \cos rx_j \cos kx_j = 0$$

The proof of (7.5) is similar.

Hence, we have

$$\frac{\partial E}{a_k} = 2\sum_{j=0}^{2m-1} y_j \cos kx_j - 2a_k \sum_{j=0}^{2m-1} \cos^2 x_j = 0$$

we must calculate

$$\sum_{j=0}^{2m-1} \cos^2 kx_j$$

which needs the following theorem.

Theorem 7.1.4. If r isn't a multiple of m, then

$$\sum_{j=0}^{2m-1} \cos^2 rx_j = m \text{ and } \sum_{j=0}^{2m-1} \sin^2 rx_j = m$$

If

$$1 \le k \le n < m$$

we have k isn't a multiple of m. By the theorem,

$$\sum_{j=0}^{2m-1} \cos^2 kx_j = m$$

Finally, we have

$$a_k = \frac{1}{m}\sum_{j=0}^{2m-1} y_j \cos kx_j, \quad k = 1, \ldots, n$$

We now need to handle the case $k = 0$. We have

$$\begin{aligned} -\frac{\partial E}{\partial a_0} &= 2\sum_{j=0}^{2m-1}\left(y_j - \frac{a_0}{2} - \sum_{r=1}^{n} a_r \cos rx_j - \sum_{r=1}^{n-1} b_r \sin rx_j\right)\frac{1}{2} \\ &= 2\sum_{j=0}^{2m-1}\left(y_j - \frac{a_0}{2}\right)\frac{1}{2} - \sum_{r=1}^{n} a_r \sum_{j=0}^{2m-1}\cos rx_j - \sum_{r=1}^{n-1} b_r \sum_{j=0}^{2m-1}\sin rx_j \qquad (7.7) \\ &= 2\sum_{j=0}^{2m-1}\left(y_j - \frac{a_0}{2}\right)\frac{1}{2} = 0 \end{aligned}$$

Note that in (7.7), we use the result from a Theorem 7.1.4. Now we have

$$\sum_{j=0}^{2m-1} y_j - ma_0 = 0$$

Since

$$\cos kx_j = 1 \text{ if } k = 0,$$

we can write a_0 as

$$a_k = \frac{1}{m}\sum_{j=0}^{2m-1} y_j \cos kx_j, \quad k = 0$$

This has the same form as the case $k = 1, \ldots, n$.

Note. Which explain that why we need to define $\frac{a_0}{2}$ instead of a_0.

Similarly, we get

$$b_k = \frac{1}{m}\sum_{j=0}^{2m-1} y_j \sin kx_j, \quad k = 1, \ldots, n-1$$

Remark. The derivation require $n < m$, which means when $k = n = m$,

$$\frac{\partial E}{\partial a_m} = 2\sum_{j=0}^{2m-1} y_j \cos mx_j - 2a_m \sum_{j=0}^{2m-1} \cos^2 mx_j = 0$$

Note that

$$x_j = -\pi + \left(\frac{j}{m}\right)\pi, \quad j = 0, 1, \ldots, 2m-1$$

Thus, we have

$$\cos^2 mx_j = \cos^2(j-m)\pi = 1$$

Then

$$a_m = \frac{1}{2m}\sum_{j=0}^{2m-1} y_j \cos mx_j,$$

is different from the general form. It has $\frac{1}{2m}$ instead of $\frac{1}{m}$.

To get the same form

$$S_n(x) = \frac{a_0 + a_n \cos nx}{2} + \sum_{k=1}^{n-1} a_k \cos kx + \sum_{k=1}^{n-1} b_k \sin kx$$

Therefore $n = m$ is also fine though earlier we used $n < m$ in our derivation.

Note. In the later discussion of the Fast Fourier Transform, we will see that the case $n = m$ is more important than the case $n < m$.

Example. Let

$$f(x) = x^4 - 3x^3 + 2x^2 - \tan(x(x-2))$$

Given $m = 5 \Rightarrow 10$ points

$$(x_j, y_j),\ j = 0, 1, \ldots, 9 \qquad x_j = \frac{j}{5},\ y_j = f(x_j)$$

Need to linearly transform these points to $[-\pi, \pi]$

$$z_j = \pi(x_j - 1)$$

Data become

$$z_j,\ f\left(1 + \frac{z_j}{\pi}\right),\ j = 0, 1, \ldots, 9$$

Let $n = 3$, least square trigonometric approximation is

$$S_3(x) = \frac{a_0}{2} + a_3 \cos 3z + \sum_{k=1}^{2}(a_k \cos kz + b_k \sin kz)$$

where

$$a_k = \frac{1}{5}\sum_{j=0}^{9} f\left(1 + \frac{z_j}{\pi}\right)\cos kz_j, \quad k = 0, 1, 2, 3$$

and

$$b_k = \frac{1}{5}\sum_{j=0}^{9} f\left(1 + \frac{z_j}{\pi}\right)\sin kz_j, \quad k = 1, 2$$

Here is the MATLAB code to calculate the coefficients

```
m = 5;
x=[0:1:2*m-1]' / 5;
y = x.^4 - 3 * x.^3 + 2 * x.^2  - tan(x .* (x-2));
z = pi * (x-1);
n = 3;
a = []; b = [];
for k=0:n
    a = [a ; 1/m * sum(y .* cos(k * z))];
end
for k=1:n-1
    b = [b; 1/m * sum(y .* sin(k * z))];
end
approx = a(1)/2 + cos(z*(1:n))*a(2:n+1) + sin(z*(1:n-1)) * b(1:n-1)
```

```
>> [y approx]
ans =
         0    0.0011
    0.4340    0.4178
    0.8981    0.9107
```

```
  1.3172    1.3208
  1.5820    1.5716
  1.5574    1.5578
  1.1980    1.2079
  0.6452    0.6414
  0.1301    0.1188
 -0.1420   -0.1278
```

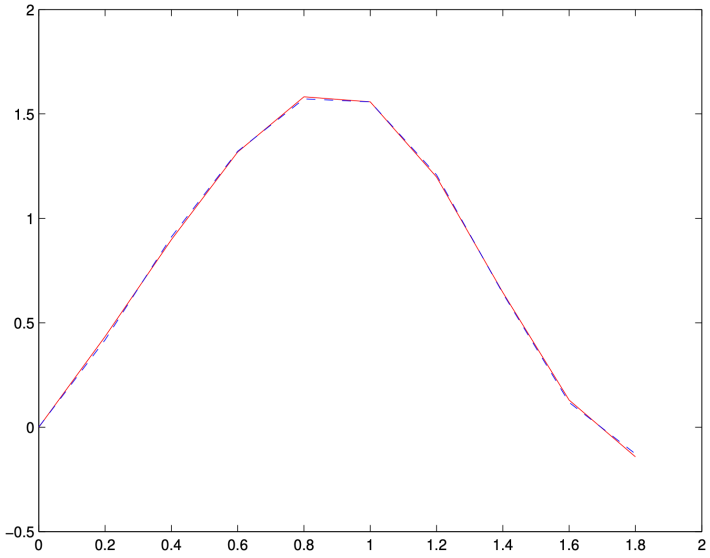


Figure 7.2: Fourier approximation of $f(x)$

Lecture 12

7.2 Fast Fourier Transform

2026.06.01

Definition 7.2.1 (Discrete Fourier Transform). Consider $n = m$. Using the coefficients obtained from the Discrete Fourier Transform (DFT) of $\{y_j\}_{j=0}^{2m-1}$, the corresponding trigonometric polynomial is given by

$$S_m(x) = \frac{a_0 + a_m \cos mx}{2} + \sum_{k=1}^{m-1} (a_k \cos kx + b_k \sin kx)$$

where

$$a_k = \frac{1}{m} \sum_{j=0}^{2m-1} y_j \cos kx_j, k = 0, 1, \ldots, n \tag{7.8}$$

and

$$b_k = \frac{1}{m} \sum_{j=0}^{2m-1} y_j \sin kx_j, k = 1, 2, \ldots, n-1 \tag{7.9}$$

This calculation requires

$$(2m)^2 \text{ multiplications and } (2m)^2 - 1 \text{ additions.}$$

which is computationally expensive for large m. The Fast Fourier Transform (FFT) is an efficient algorithm to compute the DFT, reducing the computational complexity to

$O(m \log_2 m)$ multiplications and additions.

Note. FFT is named by SIAM as one of the top 10 algorithms in the 20th century.

First, we extend the definition of DFT to a more general form. For b_k's range we use $0 \leq k \leq n$ instead of $1 \leq k \leq n-1$.

$$b_k = \frac{1}{m} \sum_{j=0}^{2m-1} y_j \sin kx_j, k = 0, 1, \ldots, m$$

as

$$\sin 0x_j = 0 \text{ and } \sin mx_j = \sin\left(m\left(-\pi + \frac{\pi j}{m}\right)\right) = 0$$

imply

$$b_0 = b_m = 0$$

Definition 7.2.2 (Complex Form of DFT)**.** Define

$$c_k = \frac{1}{2m} \sum_{j=0}^{2m-1} y_j e^{-\pi i j k/m}, \ k = 0, 1, \ldots, 2m-1 \tag{7.10}$$

We now have

$$\begin{aligned} c_k &= \sum_{j=0}^{2m-1} y_j e^{-\pi i j k/m} \\ &= \sum_{j=0}^{2m-1} y_j e^{ik(-\pi j/m + 2\pi)} = \sum_{j=0}^{2m-1} y_j e^{ik(\pi - \pi j/m + \pi)} \\ &= \sum_{j=0}^{2m-1} y_j e^{-ikx_j} e^{ik\pi} = m(a_k - ib_k)(\cos k\pi + i \sin k\pi) \\ &= m(a_k - ib_k)(-1)^k, \end{aligned}$$

where in one place we use

$$x_j = -\pi + \frac{\pi j}{m},$$

and a_k and b_k are defined in (7.8) and (7.9) respectively. If c_k is defined as in (7.10), then we can derive the following relationships:

$$\frac{c_k \cdot (-1)^k}{m} = a_k - ib_k$$

The above equation implies that

$$\begin{aligned} a_k &= \text{Re}\left(\frac{c_k(-1)^k}{m}\right) \\ b_k &= -\text{Im}\left(\frac{c_k(-1)^k}{m}\right) \end{aligned}$$

where $\text{Re}(z)$ and $\text{Im}(z)$ denote the real and imaginary parts of a complex number z, respectively.

7.2.1 Recursive Form of FFT

When $m = 2$,

$$x_j = -\pi + \frac{\pi j}{2}, j = 0, 1, 2, 3$$

$$S_2(x) = \frac{a_0 + a_2 \cos 2x}{2} + a_1 \cos x + b_1 \sin x$$

Using (7.10),

$$\begin{aligned}
c_0 &= y_0e^0 + y_1e^0 + y_2e^0 + y_3e^0,\\
c_1 &= y_0e^0 + y_1e^{-\pi i/2} + y_2e^{-\pi i} + y_3e^{-3\pi i/2},\\
c_2 &= y_0e^0 + y_1e^{-\pi i} + y_2e^{-2\pi i} + y_3e^{-3\pi i},\\
c_3 &= y_0e^0 + y_1e^{-3\pi i/2} + y_2e^{-3\pi i} + y_3e^{-9\pi i/2}
\end{aligned}$$

Note. There are some other ways to define c_k's. For example, if

$$c_k = \sum_{j=0}^{2m-1} y_j e^{\pi ijk/m},\ k = 0, 1, \ldots, 2m-1$$

then

$$\begin{aligned}
c_k &= \sum_{j=0}^{2m-1} y_j e^{ik(-\pi+\pi j/m-\pi)} = \sum_{j=0}^{2m-1} y_j e^{ikx_j} e^{-ik\pi}\\
&= m(a_k + ib_k)(\cos k\pi - i\sin k\pi)\\
&= m(a_k + ib_k)(-1)^k,
\end{aligned}$$

But, we do not consider this definition. Instead, we use the definition in (7.10) by following the common convention in the literature.

Now we need a systematic way to calculate

$$c_k, k = 0, 1, \ldots, 2m-1$$

and then find

$$a_k, b_k, k = 0, 1, \ldots, m$$

We have

$$Fy = \begin{pmatrix} \vdots \\ c_k \\ \vdots \end{pmatrix}$$

where

$$F = \begin{pmatrix}
1 & 1 & 1 & \cdots & 1\\
1 & \delta & \delta^2 & \cdots & \delta^{(2m-1)}\\
1 & \delta^2 & \delta^4 & \cdots & \delta^{2(2m-1)}\\
 & & \vdots & & \\
1 & \delta^{(2m-1)} & \delta^{2(2m-1)} & \cdots & \delta^{(2m-1)^2}
\end{pmatrix}$$

and

$$\delta = e^{-\pi i/m}$$

Note. We found that

$$e^{-\pi i/m} \text{ is the } (2m)\text{-th root of unity.}$$

because

$$\left(e^{-\pi i/m}\right)^{2m} = e^{-2\pi i} = 1$$

Now consider the case of $m = 4$

$$F = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \delta^1 & \delta^2 & \delta^3 & \delta^4 & \delta^5 & \delta^6 & \delta^7 \\ 1 & \delta^2 & \delta^4 & \delta^6 & 1 & \delta^2 & \delta^4 & \delta^6 \\ 1 & \delta^3 & \delta^6 & \delta^1 & \delta^4 & \delta^7 & \delta^2 & \delta^5 \\ 1 & \delta^4 & 1 & \delta^4 & 1 & \delta^4 & 1 & \delta^4 \\ 1 & \delta^5 & \delta^2 & \delta^7 & \delta^4 & \delta^1 & \delta^6 & \delta^3 \\ 1 & \delta^6 & \delta^4 & \delta^2 & 1 & \delta^6 & \delta^4 & \delta^2 \\ 1 & \delta^7 & \delta^6 & \delta^5 & \delta^4 & \delta^3 & \delta^2 & \delta^1 \end{pmatrix}$$

where

$$\delta = e^{-\pi i/4},\ \delta^4 = e^{-\pi i} = -1,\ \delta^8 = e^{-2\pi i} = 1$$

Then the F can be rewritten as

$$F = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \delta^1 & \delta^2 & \delta^3 & -1 & -\delta^1 & -\delta^2 & -\delta^3 \\ 1 & \delta^2 & -1 & -\delta^2 & 1 & \delta^2 & -1 & -\delta^2 \\ 1 & \delta^3 & -\delta^2 & \delta & -1 & -\delta^3 & \delta^2 & -\delta^1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\delta^1 & \delta^2 & -\delta^3 & -1 & \delta^1 & -\delta^2 & \delta^3 \\ 1 & -\delta^2 & -1 & \delta^2 & 1 & -\delta^2 & -1 & \delta^2 \\ 1 & -\delta^3 & -\delta^2 & -\delta & -1 & \delta^3 & \delta^2 & \delta^1 \end{pmatrix} \tag{7.11}$$

Suppose

$$c = \begin{pmatrix} 0 & 2 & 4 & 6 & 1 & 3 & 5 & 7 \end{pmatrix}$$

is an index permutation vector.
Then we rearrange the rows of F and the elements of y according to c:

$$Fy = F(:, c)y(c)$$

and

$$F_8(:, c) = \begin{pmatrix} F_4 & \Omega_4 F_4 \\ F_4 & -\Omega_4 F_4 \end{pmatrix}$$

We call the current form of F as F_8, then

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \delta^2 & -1 & -\delta^2 \\ 1 & -1 & 1 & -1 \\ 1 & -\delta^2 & -1 & \delta^2 \end{pmatrix}$$

and

$$\Omega_4 = \begin{pmatrix} 1 & & & \\ & \delta & & \\ & & \delta^2 & \\ & & & \delta^3 \end{pmatrix}$$

Now we have

$$\Omega_4 F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ \delta^1 & \delta^3 & -\delta^1 & -\delta^3 \\ \delta^2 & -\delta^2 & \delta^2 & -\delta^2 \\ \delta^3 & \delta^1 & -\delta^3 & -\delta^1 \end{pmatrix}$$

Then

$$Fy = F(:,c)y(c) = \begin{pmatrix} F_4 & \Omega_4 F_4 \\ F_4 & -\Omega_4 F_4 \end{pmatrix} \begin{pmatrix} y(0:2:7) \\ y(1:2:7) \end{pmatrix}$$
$$= \begin{pmatrix} I & \Omega_4 \\ I & -\Omega_4 \end{pmatrix} \begin{pmatrix} F_4 y(0:2:7) \\ F_4 y(1:2:7) \end{pmatrix}$$

We can use the same method to calculate

$$F_4 y(0:2:7) \text{ and } F_4 y(1:2:7) \tag{7.12}$$

Now we can have the recursive setting

- For $F_8 y$, multiplying with I, Ω takes $O(m)$ time since Ω is a diagonal matrix.
- Then for (7.12), we can also use $O(m)$ time, which is the same as the case of $F_8 y$.
- The number of steps is $O(\log_2 m)$.

The total time complexity is $O(m \log_2 m)$

7.2.2 Matrix-product Form of FFT

In pratice, FFT is done by matrix products rather than a recursive implementation.

Definition 7.2.3 (Matrix Factorization of FFT). Assume $m = 2^p$ for some positive integer p. Then we can write

$$F = A_t \cdots A_1 P$$

where

$$t = \log 2m$$
$$t - 1 = \log m$$
$$\vdots$$

and P is a permutation matrix

$$A_t = I_r \otimes B_L$$

where $L = 2^t$, $r = 2m/L$, and I_r is the $r \times r$ identity matrix.

> **Notation** (Kronecker product).
>
> $$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{pmatrix}$$

B_L is the $L \times L$ matrix defined as

$$B_L = \begin{pmatrix} I_{L/2} & \Omega_{L/2} \\ I_{L/2} & -\Omega_{L/2} \end{pmatrix}$$

$$\Omega_{L/2} = \text{diag}(1, \delta, \delta^2, \ldots, \delta^{L/2-1})$$

When $t = 3$

$$L = 2,\ m = 8,\ r = 1,\ I_r = 1$$

$$B_8 = \begin{pmatrix} 1 & & & 1 & & & & \\ & 1 & & & \delta & & & \\ & & 1 & & & \delta^2 & & \\ & & & 1 & & & \delta^3 & \\ 1 & & & -1 & & & & \\ & 1 & & & -\delta & & & \\ & & 1 & & & -\delta^2 & & \\ & & & 1 & & & -\delta^3 & \end{pmatrix}$$

When $t = 2$

$$L = 4,\ m = 4,\ r = 2,\ I_r = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, B_4 = \begin{pmatrix} 1 & & 1 & \\ & 1 & & \delta^2 \\ 1 & & -1 & \\ & 1 & & -\delta^2 \end{pmatrix}$$

When $t = 1$

$$L = 8,\ m = 2,\ r = 4,\ B_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

The product of $A_t \cdots A_1$ is

$$A_3 A_2 A_1 = B_8 \begin{pmatrix} B_4 & 0 \\ 0 & B_4 \end{pmatrix} \begin{pmatrix} B_2 & 0 & 0 & 0 \\ 0 & B_2 & 0 & 0 \\ 0 & 0 & B_2 & 0 \\ 0 & 0 & 0 & B_2 \end{pmatrix}$$

$$B_4 \begin{pmatrix} B_2 & 0 \\ 0 & B_2 \end{pmatrix} = \begin{pmatrix} 1 & & 1 & \\ & 1 & & \delta^2 \\ 1 & & -1 & \\ & 1 & & -\delta^2 \end{pmatrix} \begin{pmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & \delta^2 & -\delta^2 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -\delta^2 & \delta^2 \end{pmatrix}$$

Finally, we have

$$A_3 A_2 A_1 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & \delta^2 & -\delta^2 & \delta^1 & -\delta^1 & \delta^3 & -\delta^3 \\ 1 & 1 & -1 & -1 & \delta^2 & \delta^2 & -\delta^2 & -\delta^2 \\ 1 & -1 & -\delta^2 & \delta^2 & \delta^3 & -\delta^3 & \delta^1 & -\delta^1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & \delta^2 & -\delta^2 & -\delta^1 & \delta^1 & -\delta^3 & +\delta^3 \\ 1 & 1 & -1 & -1 & -\delta^2 & -\delta^2 & \delta^2 & \delta^2 \\ 1 & -1 & -\delta^2 & \delta^2 & -\delta^3 & \delta^3 & -\delta^1 & \delta^1 \end{pmatrix}$$

We can see that this is just the column rearrangement of F_8 in (7.11).

Now disscuss the permutation matrix P. We can see that

$$A_3 A_2 A_1 P,$$

where

$$P = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 1 & & & & & & & \\ & & & & 1 & & & \\ & & 1 & & & & & \\ & & & & & & 1 & \\ & 1 & & & & & & \\ & & & & & 1 & & \\ & & & 1 & & & & \\ & & & & & & & 1 \end{pmatrix} \end{array}$$

goes back to the original F

$$F = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \delta^1 & \delta^2 & \delta^3 & -1 & -\delta^1 & -\delta^2 & -\delta^3 \\ 1 & \delta^2 & -1 & -\delta^2 & 1 & \delta^2 & -1 & -\delta^2 \\ 1 & \delta^3 & -\delta^2 & \delta & -1 & -\delta^3 & \delta^2 & -\delta^1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\delta^1 & \delta^2 & -\delta^3 & -1 & \delta^1 & -\delta^2 & \delta^3 \\ 1 & -\delta^2 & -1 & \delta^2 & 1 & -\delta^2 & -1 & \delta^2 \\ 1 & -\delta^3 & -\delta^2 & -\delta & -1 & \delta^3 & \delta^2 & \delta^1 \end{pmatrix}$$

We can see that P is a permutation matrix by reversing the order of "binary digits" of indices.

0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Futher, $A_t \cdots A_1 P$ permutes the columns of $A_t \cdots A_1$. Here

$$\begin{aligned} F_8 y &= \begin{pmatrix} I & \Omega_4 \\ I & -\Omega_4 \end{pmatrix} \begin{pmatrix} F_4 & 0 \\ 0 & F_4 \end{pmatrix} \begin{pmatrix} y(0:2:7) \\ y(1:2:7) \end{pmatrix} \\ &= \begin{pmatrix} I & \Omega_4 \\ I & -\Omega_4 \end{pmatrix} \begin{pmatrix} F_4 & 0 \\ 0 & F_4 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} y \end{aligned}$$

We can see that y is rearranged according to the following order:

$$\begin{array}{cc} 0 & 000 \\ 1 & 001 \\ 2 & 010 \\ 3 & 011 \\ 4 & 100 \\ 5 & 101 \\ 6 & 110 \\ 7 & 111 \end{array} \quad \Rightarrow \quad \begin{array}{cc} 0 & 000 \\ 2 & 010 \\ 4 & 100 \\ 6 & 110 \\ 1 & 001 \\ 3 & 011 \\ 5 & 101 \\ 7 & 111 \end{array}$$

This is like that the binary representation is rotated to the left by one digit.
For example,

$$5 \Rightarrow 2(5-4)+1 = 3$$

$5-4$:	First digit "1" removed
$2(5-4)$:	Second, third digits shifted left
$2(5-4)+1$:	Add the original first digit "1" to the now last digit

Now we can see that besides the last digit, the first two digits of both the upper and the lower parts are the same:

$$\begin{array}{cc} 0 & 00 \\ 1 & 01 \\ 2 & 10 \\ 3 & 11 \end{array}$$

It is like we have a new small FFT problem

$$F_4 \tilde{y}$$

By the same setting we should have

$$F_8 y = A_3 \begin{pmatrix} I_2 & \Omega_2 & 0 & 0 \\ I_2 & -\Omega_2 & 0 & 0 \\ 0 & 0 & I_2 & \Omega_2 \\ 0 & 0 & I_2 & -\Omega_2 \end{pmatrix} \begin{pmatrix} F_2 & 0 & 0 & 0 \\ 0 & F_2 & 0 & 0 \\ 0 & 0 & F_2 & 0 \\ 0 & 0 & 0 & F_2 \end{pmatrix} \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{block diagonal}} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} y$$

We have

$$A_2 = \begin{pmatrix} I_2 & \Omega_2 & 0 & 0 \\ I_2 & -\Omega_2 & 0 & 0 \\ 0 & 0 & I_2 & \Omega_2 \\ 0 & 0 & I_2 & -\Omega_2 \end{pmatrix}$$

For the next step, we have the permutation matrix is I. Thus

$$P = I_8 \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$A_1 = \begin{pmatrix} I_1 & \Omega_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ I_1 & -\Omega_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1 & \Omega_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1 & -\Omega_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I_1 & \Omega_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & I_1 & -\Omega_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & I_1 & \Omega_1 \\ 0 & 0 & 0 & 0 & 0 & 0 & I_1 & -\Omega_1 \end{pmatrix}$$

Therefore, y is arranged by the following way

$$\begin{array}{cc} 0 & 000 \\ 1 & 001 \\ 2 & 010 \\ 3 & 011 \\ 4 & 100 \\ 5 & 101 \\ 6 & 110 \\ 7 & 111 \end{array} \quad \begin{array}{cc} 0 & 000 \\ 2 & 010 \\ 4 & 100 \\ 6 & 110 \\ 1 & 001 \\ 3 & 011 \\ 5 & 101 \\ 7 & 111 \end{array} \quad \begin{array}{cc} 0 & 000 \\ 4 & 100 \\ 2 & 010 \\ 6 & 110 \\ 1 & 001 \\ 5 & 101 \\ 3 & 011 \\ 7 & 111 \end{array}$$

$\Rightarrow$ $\Rightarrow$

Thus we have explained the reason of using the reverse of the binary digits.