

Numerical
methods
in finance



금융수치해석 소개

Lecture 01

Fall 2025
KAIST MFE
금융수치해석기법

금융에서 수치해법의 필요성 1

파생상품 가격 평가

- Black-Scholes PDE

$$\frac{\partial f}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} + rS \frac{\partial f}{\partial S} - rf = 0$$

- Analytic solution for European call option price:

$$C_0 = S_0 N(d_1) - K e^{-rT} N(d_2)$$

$$\begin{cases} d_1 = \frac{\ln(S_0/K)+(r+\sigma^2/2)T}{\sigma\sqrt{T}}, \\ d_2 = \frac{\ln(S_0/K)+(r-\sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T} \end{cases}$$

- Feynman-Kac Formula

$$C_0 = E^Q [e^{-rT}(S_T - K)^+]$$

- **해석적 해를 구할 수 없는 경우** (PDE나 다른 추가 조건이 복잡할 경우)에는 수치해법이 필요함

- Finite Difference Method(FDM)
- Numerical integration
- Monte-Carlo simulation
- Binomial or Trinomial lattices

금융에서 수치해법의 필요성 2

최적화 (Optimization)

- American option의 평가
 - 미국형 옵션의 경우 만기일 이전에 행사가 가능하므로 최적의 행사 전략(optimal exercise strategies)이 필요
 - In-the-money 상황에서 당장 행사하여 즉각적인 이익(profit)을 얻는 것과 더 나은 기회를 기다리는 것을 비교
- 포트폴리오 최적화
 - Minimum Variance Portfolio
 - Stochastic Dynamic Programming / Stochastic Control
 - Bellman Equation or HJB Equation → Algorithm trading 모델에서 활용됨
- 파라미터 estimation 또는 모델 calibration
 - Unobserved variable을 추정
 - $\min_{\alpha} \sum_j (P_j^0 - P_j(\alpha))^2$
 - 관측되는 가격과 모델 가격의 오차를 최소화

Dynamics 모델링

Differential Equation 문제

자유 낙하하는 물체의 시점 t에서 높이는?

- 운동방정식 : 높이의 시간에 대한 2계 도함수=중력가속도

$$\frac{\partial^2 h}{\partial t^2} = -g$$

- ODE(Ordinary Differential Equation)

금속막대기 특정 지점의 시점 t에서 온도는?

- 열전도방정식 : 시간 변화에 대한 온도 변화는 주변 온도에 대한 2계 도함수와 열전도율의 곱

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

- PDE(Partial Differential Equation)

금융수치해석기법의 주요 내용

Linear System of Equations / Partial Differential Equation

- 컴퓨터 산술 계산과 근사오차
- 선형방정식 해법
- 도함수 근사
- 유한차분법(Finite Difference Methods)
 - 2-Dimensional / 3-Dimensional

Optimization

- Convex 최적화
- Numerical Optimization 알고리즘
- 휴리스틱 기법
- 포트폴리오 최적화
- 계량경제모델 추정

수치해석기법에서 중요한 파트 중 하나인 Simulation은 “시뮬레이션방법론” 수업에서 별도로 다룸

컴퓨터 연산

수치연산에서 오차(error) 가능성

Truncation error

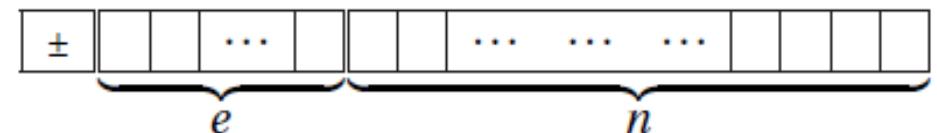
- 수학적인 모델을 단순화하면서 발생하는 오차
- 예: 도함수를 컴퓨터로 계산하기 위해서 유한차분을 하거나 연속함수를 이산화하면서 발생

$$\frac{\partial f(x)}{\partial x} \cong \frac{f(x + \epsilon) - f(x - \epsilon)}{2\epsilon}$$

- 예시: $f(x) = e^x$, $\frac{\partial f(x)}{\partial x} = e^x$
 - 수학적 해법: $f'(0) = e^0 = 1$
 - 수치적 해법:
$$\frac{e^{0.0001} - e^{-0.0001}}{2 \times 0.0001} = 1.00000000166689$$

Rounding error

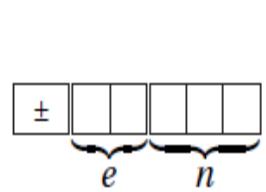
- 컴퓨터가 실수(real number)를 정확하게 표현할 수 없는 데서 기인하는 오차
- 컴퓨터에서 실수의 표현 방법 (floating point)
 - $x = \pm n \times b^e$
 - n : 가수(mantissa)
 - b : 밑(base=2)
 - e : 지수(exponent)
- ex) $91.232 = 0.71275 \times 2^7$
 $= 0.10110110011101101100100\dots \times 2^{111}$



Representation of real numbers

6 bits로 표현되는 실수 예시:

- 부호: 1bit, 지수부: 2bit, 가수부: 3bit



$e =$	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	1	0	1	1	0	1	0	1	1	0	1	0 1 2 3	$n =$	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>4</td></tr><tr><td>1</td><td>0</td><td>1</td><td>5</td></tr><tr><td>1</td><td>1</td><td>0</td><td>6</td></tr><tr><td>1</td><td>1</td><td>1</td><td>7</td></tr></table>	1	0	0	4	1	0	1	5	1	1	0	6	1	1	1	7	4 5 6 7
0	0	0	0																																		
0	1	0	1																																		
1	0	1	0																																		
1	1	0	1																																		
1	0	0	4																																		
1	0	1	5																																		
1	1	0	6																																		
1	1	1	7																																		

- 표현 가능한 실수는 균등한 간격을 이루지 않음
- 큰 숫자일수록 표현 가능한 숫자간 간격이 넓어짐
- 컴퓨터 계산에서 실수를 표현할 때는 이와 같은 제한을 염두에 두고, 필요에 따라 정밀도를 조정하는 것이 중요

- $x = \pm n \times b^{e-3}$

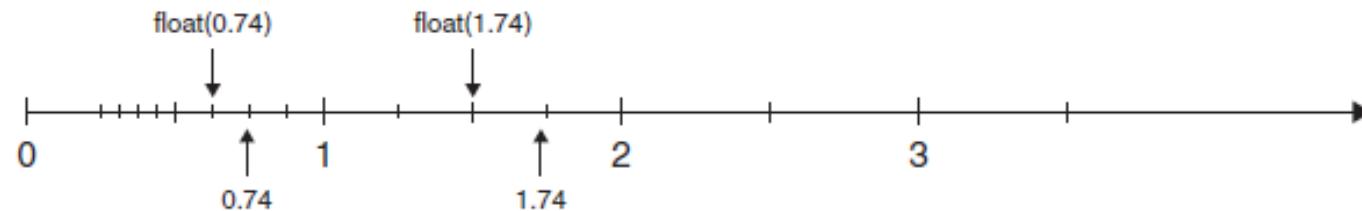
n	2^{e-3}	$e = -1$	$e = 0$	$e = 1$	$e = 2$			
		$1/16$	$1/8$	$1/4$	$1/2$			
<table border="1"><tr><td>1</td><td>0</td><td>0</td></tr></table>	1	0	0	$=2^2=4$	$1/4$	$1/2$	1	2
1	0	0						
<table border="1"><tr><td>1</td><td>0</td><td>1</td></tr></table>	1	0	1	$=2^2+2^0=5$	$5/16$	$5/8$	$5/4$	$5/2$
1	0	1						
<table border="1"><tr><td>1</td><td>1</td><td>0</td></tr></table>	1	1	0	$=2^2+2^1=6$	$3/8$	$3/4$	$3/2$	3
1	1	0						
<table border="1"><tr><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	$=2^2+2^1+2^0=7$	$7/16$	$7/8$	$7/4$	$7/2$
1	1	1						



float 연산

float(·) 연산자 – floating point 연산의 값을 의미

- 앞의 예에서 $\text{float}(1.74) = 1.5$ 이고, $\text{float}(0.74) = 0.625$ 임



- 따라서 $\text{float}(1.74 - 0.74) = 0.875$ 임
 - 정확한 결과인 1이 floating point로 표현됨에도 불구하고 연산 결과는 다른 값이 될 수 있음

Precision of floating point arithmetic

Machine Epsilon

- $\text{float}(1 + \epsilon_{\text{mach}}) > 1$ 을 만족하는 가장 작은 양수 ϵ_{mach} 를 의미함

```
1: e = 1
2: while 1 + e > 1 do
3:   e = e / 2
4: end while
5: εmach = 2e
```

- $\epsilon_{\text{mach}} = 2.220446049250313 \times 10^{-16}$

```
print(np.finfo(float).eps)
# 2.22044604925e-16

print(np.finfo(np.float32).eps)
# 1.19209e-07
```

- $\text{float}(\text{float}(1 + e) + e) = 1,$
- $\text{float}(1 + \text{float}(e + e)) > 1.$

Double precision

Double 정밀도: 64bit로 실수를 표현함

- 부호: 1bit, 가수: 52bit, 지수: 11bit
- 지수의 범위: -1,023 ~ 1,024
- 표현 가능한 실수의 범위: $m < f < M$
- $m \approx 2.22 \times 10^{-308}$ and $M \approx 1.79 \times 10^{308}$
 - 관측 가능한 우주의 원자의 수는 대략 10^{80} 정도로 추정되는 점을 감안하면 double 정밀도로 표현 가능한 수의 범위는 상당히 어마어마한 크기임
 - 표현할 수 있는 숫자의 개수는 $2^{64} = 18,446,744,073,709,551,616$ 개
(약 1,844경 6,744조 개)

- 수식의 결과가 m 보다 작은 경우에는 underflow가 발생하여 결과가 0이 됨
- 반대로 M 보다 큰 경우에는 overflow가 발생하여 결과가 inf가 되어 이후 계산을 망치게 됨

Floating point arithmetic 계산 한계

$$\sum_{k=1}^{\infty} \frac{1}{k} = \infty$$

- 컴퓨터로 위의 합계를 계산하면 값이 무한히 증가할까?
- Floating point 계산 한계로 인해 더 이상 증가하지 않는 지점이 발생함
- Underflow로 인해 $1/k$ 가 0이 되거나 overflow로 합계 값이 M 보다 커지기 이전에 값이 더 이상 증가하지 않게됨 → Why?

계산 오차

- 절대오차(Absolute Error)

- 절대 오차는 정확한 값 x 와 근사값 \bar{x} 사이의 절대 차이로 정의
- x 값의 크기에 따라 오차의 의미가 달라짐

$$e_{abs} = |x - \bar{x}|$$

- 상대오차(Relative Error)

- 상대 오차는 정확한 값의 크기와 비교하여 오류를 측정할 때 유용

$$e_{rel} = \frac{|x - \bar{x}|}{|x|}$$

- x 값이 0이거나 0에 매우 가까운 값인 경우 사용하기 어려움

- 절대오차와 상대오차를 결합한 오차

$$e_{comb} = \frac{|x - \bar{x}|}{|x| + 1}$$

- x 가 0에 가까운 값인 경우 절대오차에 근사하고, $|x|$ 절대값이 커지면 상대오차에 근사함

유한차분을 이용한 도함수의 근사법

도함수의 정의

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}.$$

- h 의 값이 0으로 수렴하는 대신에 상당히 “작은” 값으로 대체하고, $f'(x)$ 를 근사함

- h 의 값을 어떻게 정해야 좋은 근사값을 얻을 수 있을까? Truncation error를 최소화하기 위해서 최대한 작은 값 선택
 - 그러나 너무 작은 값을 선택하면 $\text{float}(x + h) = \text{float}(x)$ 이 되거나,
 - 그렇지 않더라도 $\text{float}(f(x + h)) = \text{float}(f(x))$ 가 될 수 있음

테일러 확장

- 함수 f 에 대한 테일러 확장식

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n(x + h)$$

$$R_n(x + h) = \frac{h^{n+1}}{(n+1)!}f^{(n+1)}(\xi), \quad \xi \in [x, x + h],$$

1계 도함수의 근사

- Taylor's expansion up to order two:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi) \quad \text{with } \xi \in [x, x+h],$$

Forward difference approximation:

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

Backward difference approximation:

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

Central difference approximation:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(\xi),$$

Central difference approximation

$$\begin{cases} f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\xi_+) & \text{with } \xi_+ \in [x, x+h], \\ f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\xi_-) & \text{with } \xi_- \in [x-h, x], \end{cases}$$

이 두 수식을 빼면

$$\Rightarrow f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{6}(f'''(\xi_+) + f'''(\xi_-))$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(\xi),$$

Central difference 방식의 truncation error의 order가 $O(h^2)$ 로 다른 방법에 비해 오차가 줄어들

2계 도함수와 partial derivatives의 근사

2계 도함수

$$\begin{cases} f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(\xi_+) \\ f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(\xi_-) \end{cases}$$

이 두식을 더하면,

$$\Rightarrow f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h^2}{24}f''''(\xi)$$

Partial derivatives:

$$f_x = \frac{f(x+h_x, y) - f(x-h_x, y)}{2h_x}$$

Cross partial derivatives:

$$f_{xy} = \frac{\frac{f(x+h_x, y+h_y) - f(x-h_x, y+h_y)}{2h_x} - \frac{f(x+h_x, y-h_y) - f(x-h_x, y-h_y)}{2h_x}}{2h_y}$$

$$= \frac{1}{4h_x h_y} (f(x+h_x, y+h_y) - f(x-h_x, y+h_y) - f(x+h_x, y-h_y) + f(x-h_x, y-h_y)).$$

Truncation error와 Rounding error

Forward difference의 경우

truncation error bound:

- Truncation error: $-\frac{h}{2} f''(\xi)$
- 2계도함수의 절대값이 어떤 M 이라는 값보다 작다면, $|f''(t)| \leq M$
- $|f'_h(x) - f'(x)| = |\frac{h}{2} f''(x)| \leq \frac{h}{2} M \implies h$ 가 작을수록 오차가 작아짐

Rounding error:

- 반올림오차: $|float(f(x)) - f(x)| \leq \epsilon$
- $|float(f'_h(x)) - f'_h(x)| \leq \frac{2\epsilon}{h}$

총오차:

- $g(h) = |float(f'_h(x)) - f'(x)| \leq \frac{2\epsilon}{h} + \frac{M}{2}h$
- 총오차를 최소화하기 위한 최적의 $h^* = \sqrt{\frac{4\epsilon}{M}}$.
 $\epsilon = \epsilon_{mach}$ & $M = 1$ 일 때 $\approx 2.8 \times 10^{-8}$

Central difference인 경우:

$$g(h) = \frac{2\epsilon}{h} + \frac{h^2}{3}M$$
$$\implies h^* = \sqrt[3]{\frac{3\epsilon}{M}} \approx 0.843 \times 10^{-5}$$

h 에 따른 영향 예시

- 다음의 함수 $f(x)$ 에 대한 미분계수를 수치해법으로 계산하는 경우

$$f(x) = \cos(x^x) - \sin(e^x)$$

$$f'(x) = -\sin(x^x)x^x(\log x + 1) - \cos(e^x)e^x$$

- $x = 1.5$ 에서 도함수의 수치적 근사의 상대 오차 그래프. h 의 값이 10^{-16} 에서 1까지 변화
 - Central difference 근사의 최적 h 값은 10^{-5} 근처에서 최소 오차를 보임
 - Forward difference 근사의 최적 h 값은 10^{-8} 근처에서 최소 오차를 보임

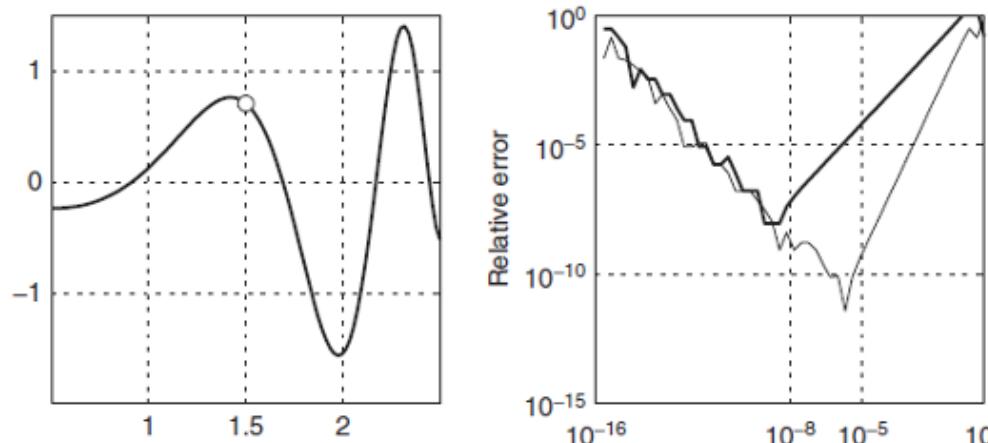


Figure 2.1 Function $f(x) = \cos(x^x) - \sin(e^x)$ (left panel) and relative error of the numerical approximation of the derivative at $x = 1.5$ for forward difference (thick line) and central difference (thin line).

수치적 불안정성과 악조건 문제

수치적 불안정성 (Numerical Instability)

- 알고리즘이 반올림 오차를 크게 증폭시키는 상황
- 초기 반올림 오차가 작더라도 알고리즘에 의해 크게 부풀려져 결과가 부정확해질 수 있음

악조건 문제 (III-Conditioned Problem)

- 입력 데이터의 작은 변동이 출력 솔루션에 큰 변화를 일으키는 문제
- 이는 문제가 입력 데이터의 정확성에 매우 민감하다는 것을 의미

수치적 불안정성과 악조건 문제 구별

- **수치적 불안정성**은 **알고리즘**의 속성 → 더 안정적인 알고리즘 사용으로 완화 가능
 - 이는 더 안정적인 알고리즘을 사용하거나 수치적 정밀도를 개선하여 완화할 수 있는 문제임
 - 알고리즘을 선택하거나 설계할 때 오류가 알고리즘의 단계에서 어떻게 전파되는지 고려해야 함
- **악조건 문제**는 **문제 자체**의 속성 → 알고리즘을 변경한다고 해서 완전히 제거할 수는 없으며, 입력 데이터의 정확성을 개선하고 정규화 기법을 사용하는 것이 필요함
 - 민감성을 처리하도록 설계된 알고리즘을 적용하는 것이 중요

수치적 불안정성 알고리즘 예시

- 이차방정식 $ax^2 + bx + c = 0$ 의 해 $\Rightarrow x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$, $x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$
- $a = 1, c = 2$ 일 때, b 값에 따른 계산 결과(floating point에서 5자리 정밀도를 가진다고 가정함)
- 계산 알고리즘 1



1: $\Delta = \sqrt{b^2 - 4ac}$	2: $x_1 = (-b - \Delta)/(2a)$	3: $x_2 = (-b + \Delta)/(2a)$	b	Δ	float(Δ)	float(x_2)	x_2	$\frac{ float(x_2) - x_2 }{ x_2 +1}$
5.2123	4.378135	4.3781	-0.41708	-0.4170822	1.55×10^{-6}			
121.23	121.197	121.20	-0.01500	-0.0164998	1.47×10^{-3}			
1212.3	1212.2967	1212.3	0	-0.001649757	Catastrophic cancellation			

Note that $\text{float}(x_2) = (-b + \text{float}(\Delta))/(2a)$

- 계산 알고리즘 변경



1: $\Delta = \sqrt{b^2 - 4ac}$	2: if $b < 0$ then	3: $x_1 = (-b + \Delta)/(2a)$	4: else	5: $x_1 = (-b - \Delta)/(2a)$	6: end if	7: $x_2 = c/(ax_1)$	b	float(Δ)	float(x_1)	float(x_2)	x_2
1212.3							1212.3	1212.3	-1212.3	-0.0016498	-0.001649757

약조건 문제 예시

선형시스템 $Ax = b$ 의 해

$$A = \begin{bmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{bmatrix} \quad b = \begin{bmatrix} 0.217 \\ 0.254 \end{bmatrix}.$$

- 이 경우 정확한 해는 $x = [1, -1]'$ 임

```
import numpy as np
# Define matrix A
A = np.array([[0.780, 0.563],
              [0.913, 0.659]])
# Define vector b
b = np.array([[0.217], [0.254]])
# Solve for x in Ax = b
x = np.linalg.inv(A).dot(b)
print(x)
# [[ 1.], [-1.]]
```

입력 파라미터를 약간 변경할 때,

$$E = \begin{bmatrix} 0.001 & 0.001 \\ -0.002 & -0.001 \end{bmatrix}$$

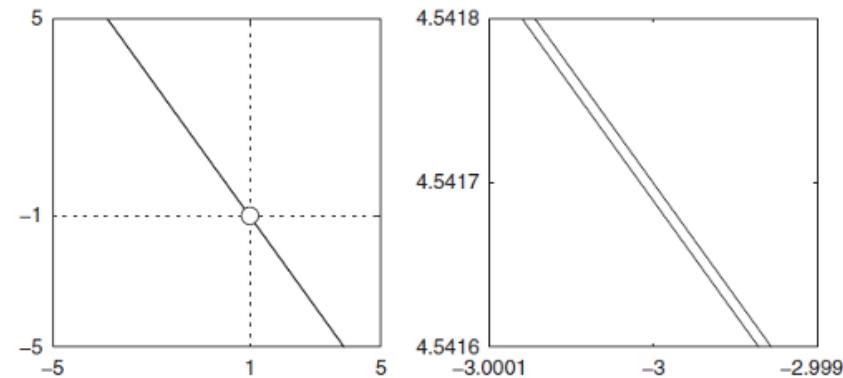
- $(A + E)x_e = b$ 의 해는?

```
# Create a small perturbation matrix E
E = np.array([[0.001, 0.001],
              [-0.002, -0.001]])
# Solve for xe in (A+E)x = b
xe = np.linalg.inv(A + E).dot(b)
print(xe)
# [[-5.          ], [ 7.30851064]]
```

악조건 문제 예시

행렬 A 에 매우 작은 변화를 주었음에도 불구하고, 해가 상당히 변함

- 선형 시스템의 해는 두 직선의 교점에 해당



- 왼쪽 그림에서는 두 직선이 매우 가까워서 거의 평행해 보임
- 오른쪽 그림은 좌표 $(-3, 4.54)$ 주변을 확대하여 두 개의 명확히 구별되는 직선을 보여줌
- 따라서 계수의 작은 변화가 교점의 위치에 큰 영향을 미칠 수 있음

문제 해결

- 이런 문제를 해결하려면 데이터를 정밀하게 측정하고,
- 가능한 한 오류를 줄이기 위해 적절한 알고리즘을 사용하는 것이 중요

행렬의 조건수(condition number)

조건수

- 문제 $f(x)$ 의 해가 데이터 x 의 변화에 얼마나 민감한지를 나타내는 값
- 탄력성의 절대값(absolute value of the elasticity)

$$\text{cond}(f(x)) \approx \frac{|xf'(x)|}{|f(x)|}.$$

- 탄력성이 크다면 문제는 악조건(ill-conditioned)임
- 탄력성이 큰 경우:** $f'(x)$ 가 매우 큰 경우, x 가 매우 큰 경우, $f(x)$ 가 매우 작은 경우

선형시스템의 조건수

- 일반적으로 문제의 조건수를 추정하는 것은 어려움
- 하지만, 선형 시스템에서는 계수 행렬 A 의 조건수를 다음과 같이 정의함

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

- 일반적으로 $\text{cond}(A) > 1/\sqrt{\text{eps}}$ 인 경우 수치 결과에 대해서 우려해야 함 ($1/\sqrt{\text{eps}} \approx 6.7 \times 10^7$)
- 더 작은 조건수에서도 문제의 종류에 따라 의심스러워질 수 있음

알고리즘의 계산 복잡도(computational complexity)

계산복잡도

- 계산 복잡도 분석은 알고리즘의 효율성을 측정하는 도구로, 알고리즘의 성능을 비교하고 평가하는 데 사용

문제 크기(problem size)

- 알고리즘의 계산 노력의 크기를 결정
- 일반적으로 문제 크기는 알고리즘이 처리해야 할 요소의 수(데이터의 양)로 표현
- 선형 시스템 $Ax = b$ 의 해를 구하는 경우, 문제의 크기는 행렬 A 의 차수 n 으로 정의
- 행렬 곱셈의 경우 문제 크기는 두 행렬의 차원으로 정의

알고리즘 비교의 기준

- 실행 시간:** 가장 중요한 비교 기준
 - 실행 시간 측정이 쉬워야하고, 계산 플랫폼에 독립적이어야 하므로 기본 연산의 수로 측정됨
 - 알고리즘의 복잡도는 크기 n 의 문제를 해결하는데 필요한 기본 연산의 수로 정의
- 공간복잡도:** 알고리즘 실행에 필요한 고속 메모리의 양
- 코드의 단순성:** 코드의 이해 및 유지보수 용이성
- 컴퓨터 성능 발전으로 공간복잡도가 문제가 되는 경우는 줄어들고 있음

알고리즘 복잡도

복잡도 측정방법

- 일반적으로 덧셈, 뺄셈, 곱셈, 나눗셈과 같은 기본 연산의 수를 세는 방식으로 복잡도를 측정 (flop, floating point operation).

복잡도 예시

- 두 알고리즘 A_1 과 A_2 가 있는데, A_1 의 연산 수는 $C_{A_1}(n) = \frac{1}{2}n^2$ 이고, A_2 의 연산 수는 $C_{A_2}(n) = 5n$ 이라고 가정

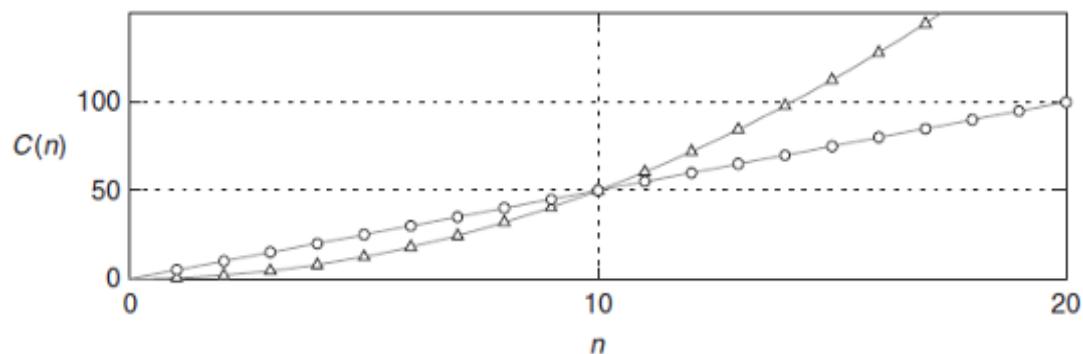


Figure 2.3 Operation count for algorithm A_1 (triangles) and A_2 (circles).

- $n > 10$ 에서는 A_1 보다 A_2 의 연산이 빠름
- n 값이 큰 상황에서 계산 시간이 중요해지므로 복잡도를 결정하는 것은 연산 수의 n 에 대한 order임
- 알고리즘 A_1 의 복잡도는 $O(n^2)$
- 알고리즘 A_2 의 볍잡도는 $O(n)$
- $\frac{1}{3}n^3 + n^2 + \frac{2}{3}n$ 의 연산 수를 갖는 알고리즘의 복잡도는 $O(n^3)$ 임

알고리즘 효율성

알고리즘의 분류

- 알고리즘은 연산 수 함수의 순서에 따라 두 가지 주요 그룹으로 분류됨
 1. **다항 시간 알고리즘(Polynomial time algorithms)**: 연산 수가 문제 크기의 다항 함수로 표현
 2. **비다항 시간 알고리즘(Non-polynomial time algorithms)**: 다항 함수로 표현되지 않는 알고리즘

컴퓨터 성능과 알고리즘 효율성

- 컴퓨터 성능은 초당 플롭 수로 측정됨
- 개인용 컴퓨터는 1980년대 초반 수 Kflops(10^3 flops)에서 2000년대 말에는 수 Gflops(10^9 flops)로 발전
- 사용 가능한 빠른 메모리의 크기도 같은 비율로 증가
- 비다항 시간 알고리즘의 비효율성은 빠른 컴퓨터로도 해결되지 않음
- 2^n 인 알고리즘의 경우 주어진 시간 내에 해결할 수 있는 최대 문제 크기를 N 이라고 하면, 계산 속도를 1,024배 늘리더라도 해결할 수 있는 문제 크기는 $N + 10$ 에 불과함

기본 선형 대수 연산의 연산 수

- 기본 연산 (덧셈, 뺄셈, 곱셈, 나눗셈) 각각은 하나의 플롭(flop)으로 계산
- 벡터 $x, y \in \mathbb{R}^n, z \in \mathbb{R}^m$ 와 행렬 $A \in \mathbb{R}^{m \times r}, B \in \mathbb{R}^{r \times n}$ 에 대해서 각 연산에 필요한 플롭 수

연산	플롭 수
$x + y$	n
$x^T x$	$2n \ (n > 1)$
xz^T	nm
AB	$2mrn \ (r > 1)$

- 벡터 덧셈 $x + y$: 총 n 번의 덧셈이 필요
- 벡터 내적 $x^T x$: 총 $2n$ 번의 연산(곱셈 n 번, 덧셈 $n - 1$ 번)으로 계산됨
- 벡터 외적 xz^T : 총 nm 번의 곱셈이 필요
- 행렬 곱셈 AB : 총 $2mrn$ 번의 연산이 필요