

Numerical  
methods  
in finance



# 선형방정식 및 최소자승법

## Lecture 02

Fall 2025  
KAIST MFE  
금융수치해석기법

# 선형 방정식 시스템(Systems of linear equations)

- 수치 문제에서 매우 자주 발생하며, 따라서 효율적인 해결 방법의 선택은 매우 중요
- 문제는 행렬  $A$ , 오른쪽 항 벡터  $b$ , 미지수 벡터  $x$ 가 주어졌을 때  $Ax = b$  의 해를 찾는 것으로 구성

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \iff \left[ \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array} \right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

## 선형 방정식 시스템

- 위의 시스템에서 해가 존재하고 유일하려면  $A$ 의 행렬식(determinant)이 0이 아니어야 함 ( $\det(A) \neq 0$ )
- $x = A^{-1}b$ 는 해를 찾기 위해  $A^{-1}$  를 계산해야 한다고 생각할 수 있지만, 효율적인 수치적 방법은  $A^{-1}$  를 계산하거나  $\det(A)$ 를 계산하여 해의 존재를 확인하지 않음
- 가우스-조던 방법 계산 복잡도:  $O(2n^3)$

## 수치적 해법

- 가우스 소거법
- LU 분해
- QR 분해

문제의 크기와 특성에 따라 적절한 알고리즘을 선택하는 것이 성능에 큰 영향을 미침

# 선형 방정식 문제 해법 소개

---

## 선형 시스템을 해결하기 위한 수치적 방법의 두 가지 주요 범주

**직접 방법(Direct Methods)**: 행렬을 분해하는 방법을 사용

- **가우스 소거법 (Gaussian Elimination)**: 행렬을 상삼각 행렬로 변환한 후, 역대입법(back substitution)으로 해를 구함
- **LU 분해 (LU Decomposition)**: 행렬  $A$ 를 하삼각 행렬  $L$ 과 상삼각 행렬  $U$ 로 분해하여,  $Ly = b$ 와  $Ux = y$ 를 차례로 풀어 해를 구함
- **Cholesky 분해 (Cholesky Decomposition)**: positive definite 행렬  $A$ 를  $A = LL^T$  형태로 분해하여 해를 구함
- **QR 분해 (QR Decomposition)**: 행렬  $A$ 를 직교 행렬  $Q$ 와 상삼각 행렬  $R$ 로 분해하여,  $Rx = Q^Tb$ 를 풀어 해를 구함

**반복 방법(Iterative Methods)**: stationary 및 non-stationary 반복 방법

- **Jacobi 방법**: 각 반복 단계에서  $x$ 의 새로운 값을 독립적으로 업데이트
- **Gauss-Seidel 방법**: 각 반복 단계에서  $x$ 의 새로운 값을 바로 반영하여 다음 계산에 사용
- **SOR 방법 (Successive Over-Relaxation)**: Gauss-Seidel 방법을 확장하여 수렴 속도를 향상
- **CG 방법 (Conjugate Gradient)**: positive definite 대칭 행렬에 대해 빠르게 수렴
- **GMRES 방법 (Generalized Minimal Residual)**: 비정지형 반복 방법으로, 일반적인 비대칭 행렬에 대해 사용

# 선형방정식 문제의 해결 방법

---

## 실전 응용에서의 고려 사항

- 실제 응용에서는 행렬의 특성과 문제의 크기에 따라 적절한 방법을 선택하는 것이 중요
- 직접 방법은 일반적으로 작고 조밀한 행렬에 적합하며, 반복 방법은 큰 규모의 희소 행렬에 더 적합
- 문제의 특성을 잘 파악하고, 행렬의 구조와 특성을 최대한 활용하여 계산 효율성을 높이는 것이 중요

## 방법 선택의 기준

- **조밀한 행렬(Dense Matrix)**
  - 거의 모든 요소가 0이 아닌 경우
  - 직접 방법이 일반적으로 더 효율적
- **희소 행렬(Sparse Matrix)**
  - 행렬의  $n^2$  요소 중 극히 일부만이 0이 아닌 경우
  - 반복 방법이 일반적으로 더 효율적
- **띠 행렬(Banded Matrix)**
  - 행렬의 0이 아닌 요소가 대각선 근처에 집중되어 있는 경우
  - 밴드 구조를 이용한 직접 방법 (예: 밴드 LU 분해)이나 반복 방법 모두 사용 가능
- **특정 구조 (삼각 행렬(triangular), 대칭 행렬 등)**
  - 행렬의 특성을 활용한 특별한 직접 방법이 사용될 수 있음

# 직접 방법(Direct Methods)

---

- 직접방법은 원문제를 풀기 쉬운 형태의 문제로 변환함
- 예를 들어,  $Ax = b$ 를  $Ux = c$  ( $U$ 는 삼각행렬 또는 대각행렬)  $Qx = c$  ( $Q$ 는 직교 orthogonal 행렬) 또는 의 형태로 바꿈

## 삼각시스템(Triangular System)

- 삼각 시스템은 선형 방정식의 계수 행렬이 상삼각 행렬 또는 하삼각 행렬 형태를 가지는 경우

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

- 역행렬을 구하지 않고 손쉽게 해를 구할 수 있음 ( $l_{11}, l_{22} \neq 0$  가정)

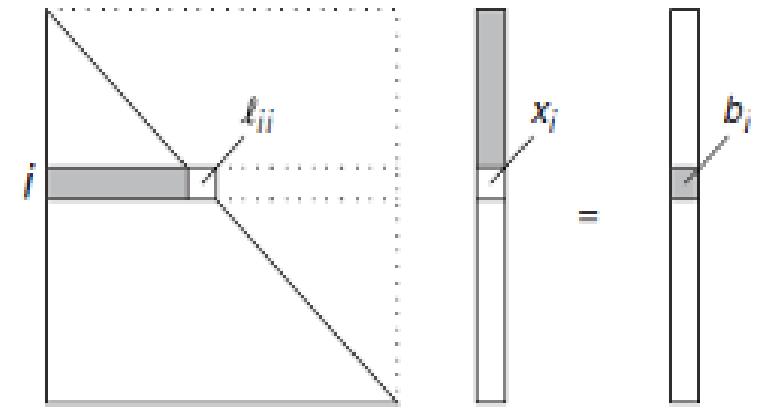
$$x_1 = b_1/l_{11}$$
$$x_2 = (b_2 - l_{21}x_1)/l_{22}.$$

# Forward-substitution / Back-substitution

## Forward-substitution

- 선형시스템  $Lx = b$ 에서  $L$  행렬이 하삼각행렬(lower triangular matrix)인 경우

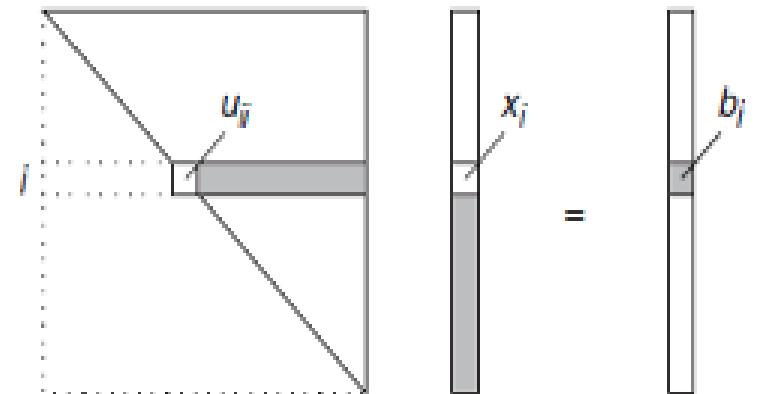
$$x_i = \left( b_i - \sum_{j=1}^{i-1} \ell_{ij} x_j \right) / \ell_{ii}$$



## Back-substitution

- 선형시스템  $Ux = b$ 에서  $U$  행렬이 상삼각행렬(upper triangular matrix)인 경우

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



# 알고리즘

---

---

## Algorithm 1 Forward substitution.

---

```
1:  $b_1 = b_1 / L_{1,1}$ 
2: for  $i = 2 : n$  do
3:    $b_i = (b_i - L_{i,1:i-1} b_{1:i-1}) / L_{ii}$ 
4: end for
```

---

$x$ 벡터의 크기가  $n$ 일 때, 플롭수?

$$\sum_{i=1}^n (2i - 1) = n^2$$

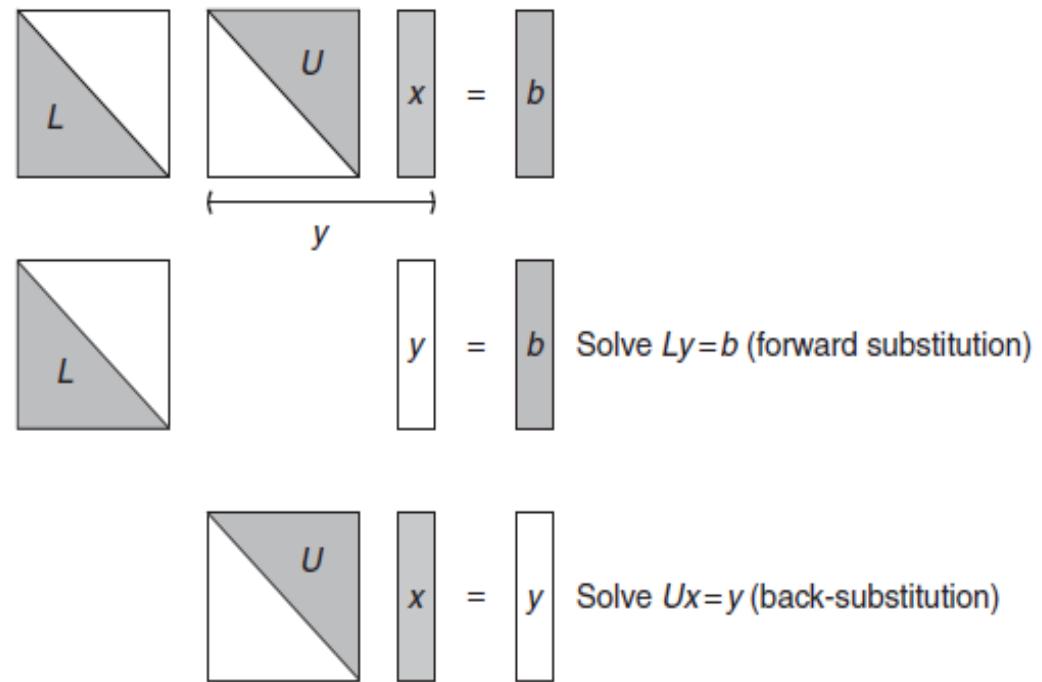
$$O(n^2)$$

# LU 분해 (LU factorization)

## LU 분해

### 가우스소거법을 행렬로 표현한 방법

- LU 분해는 행렬  $A$ 가 밀집(dense)하고 특별한 구조가 없을 때 선택할 수 있음
- 행렬  $A$ 를 하삼각 행렬  $L$  과 상삼각 행렬  $U$ 의 곱으로 분해
- LU 분해의 계산 복잡도:  $O\left(\frac{2}{3}n^3\right)$ 
  - 역행렬 계산보다 3배 정도 빠름
- $Ax = b$ 에서  $A$ 를  $LU$ 로 대체하여 변환된 시스템  $LUx = b$ 를 두 단계로 해를 구함
- 각 단계는 삼각 시스템을 푸는 과정을 포함. 이는 다음 그림과 같이 중간 벡터  $y$ 를 사용하여 설명



# LUP 분해

## LUP 분해

- 행렬의 성분들을 적절하게 정렬하여 LU분해가 가능하도록 함
- 행 또는 열 교환(pivoting)이 필요한 경우 치환행렬 P를 추가하여 LUP 분해라고 함
- $PA = LU$
- 모든 정방행렬 A는 LUP 분해 가능함

## 선형방정식 풀이

- $Ax = b$
- $PAx = Pb$
- $LUx = Pb$
- $Ly = Pb$ 를  $y$ 에 대해서 풀고,  $Ux = y$ 를  $x$ 에 대해서 푼다.

```
import numpy as np
from scipy.linalg import lu

# 예제 행렬 A와 벡터 b 정의
A = np.array([[3, 1, 6],
              [2, 1, 4],
              [1, 4, 5]], dtype=float)
b = np.array([12, 10, 8], dtype=float)

# LU 분해 수행
P, L, U = lu(A)

# 전방 대입 (Ly = Pb)
Pb = np.dot(P, b)
y = np.zeros_like(b)

for i in range(len(y)):
    y[i] = Pb[i] - np.dot(L[i, :i], y[:i])

# 후방 대입 (Ux = y)
x = np.zeros_like(y)

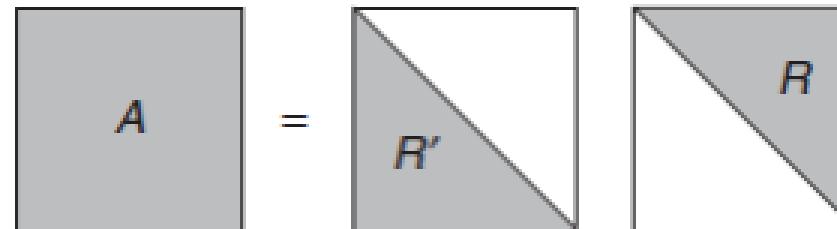
for i in range(len(x)-1, -1, -1):
    x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]

# 결과 출력
print("Solution x:")
print(x)
```

# 촐레스키 분해(Cholesky factorization)

## 촐레스키 분해를 이용한 방법

- 행렬  $A$ 가 대칭(symmetric)이고 양정부호(positive-definite)이면 촐레스키 분해가 가능함
  - Symmetric:  $A = A^T$
  - Positive-definite:  $x^T A x > 0$ , for all non-zero vector  $x$
- $A = R'R$ ,  $R$ 은 상삼각행렬
- LU 분해보다 연산량이 적으며, 대칭적이고 양정부호인 행렬에 적합
- 촐레스키 분해 계산 복잡도:  $\frac{1}{3}O(n^3)$  → LU분해에 비해 2배 가량 빠름


$$A = R'R$$

- $A$ 를  $R'R$ 로 대체하면 앞서 LU 분해를 이용한 해법과 마찬가지로 forward / back-substitution 방법을 이용해서 쉽게 방정식의 해를 찾음
- 파이썬 numpy를 이용한 촐레스키 분해: `np.linalg.cholesky(A)`

# 촐레스키분해를 통한 다변량 정규분포 생성

## Cholesky 분해는 다변량 정규분포를 생성하는 데도 사용

- 공분산행렬  $\Sigma$  를 가지는 다변량 정규분포를 생성하는 경우
  - 공분산 행렬  $\Sigma$ 의 Cholesky 분해를 수행하여  $R$ 을 구한다.
  - $R$ 과 독립적인 표준 정규분포를 따르는 난수 벡터  $u$ 를 곱하여 원하는 분포의 난수 벡터  $x$ 를 생성한다.
  - $x = R'u, \quad u \sim N(0, I)$
- $x$ 의 공분산행렬을 계산하면,

$$\begin{aligned} E(xx') &= E(R' \underbrace{uu'}_I R) \\ &= R'IR = R'R = \Sigma. \end{aligned}$$

```
import numpy as np

#공분산 행렬 Sigma 생성
Sigma = np.array([[4, 2], [2, 3]])
R = np.linalg.cholesky(Sigma)

#표준 정규분포를 따르는 난수 벡터 u 생성
u = np.random.normal(size=(2, 1000))

#다변량 정규분포를 따르는 난수 벡터 x 생성
x = np.dot(R, u)

#결과 검증: x의 공분산 행렬은 Sigma에 근접해야 함
print("생성된 x의 공분산 행렬:\n", np.cov(x))
```

# QR 분해 (QR decomposition)

## QR 분해

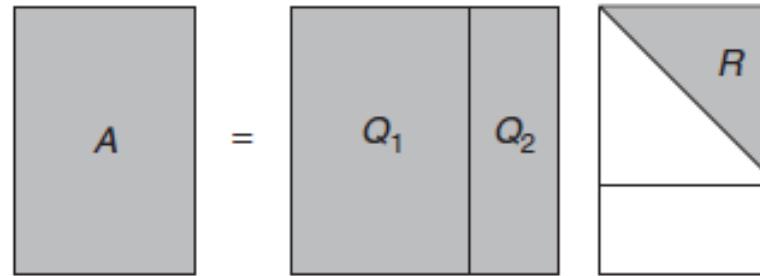
- $A \in R^{m \times n}$ 이 full column rank인 경우, QR 분해가 가능함
- Q는 직교행렬(orthogonal matrix)임. 즉,  $Q'Q = I$
- A행렬이 정방행렬인 경우  $Q_2$ 는 사라지고 R은 삼각행렬임
- LU분해보다 계산량이 많음

## 선형방정식의 해법

$$Q \begin{pmatrix} R \\ \vdots \end{pmatrix} \begin{pmatrix} x \\ \vdots \end{pmatrix} = b$$

$$\begin{pmatrix} R \\ \vdots \end{pmatrix} \begin{pmatrix} x \\ \vdots \end{pmatrix} = Q' \begin{pmatrix} b \\ \vdots \end{pmatrix}$$

Solve  $Rx = Q'b$   
(back-substitution)



```
# QR 분해 수행  
Q, R = np.linalg.qr(A)
```

```
# Q와 R 확인  
print("Q 행렬:\n", Q)  
print("R 행렬:\n", R)
```

```
# Q^T * b 계산  
Q_T_b = np.dot(Q.T, b)
```

# 특이값 분해 (Singular value decomposition)

## 특이값 분해 – SVD

- 행렬을 3개의 특수 행렬의 곱으로 분해
- 악조건 문제 회피: full-column rank 가 아닌 행렬도 안정적으로 분해
- 가장 계산량이 많은 행렬 분해 방법
- 행렬  $A \in R^{m \times n}$ 은 두 개의 직교행렬  $U \in R^{m \times m}$  과  $V \in R^{n \times n}$ , 그리고  $\Sigma = diag(\sigma_1, \sigma_2, \dots, \sigma_p) \in R^{m \times n}$ ,  $p = \min(m, n)$
- $\sigma_i$  - singular values라고 부르며,  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p \geq 0$ 의 관계를 만족함
- 행렬의 rank를 계산하거나 차원 축소에 사용 가능

## SVD를 이용한 선형 방정식 해법

1.  $Ax = b$ 를  $U\Sigma V'x = b$ 로 변형
2. 양변에  $U'$  을 곱하여  $\Sigma V'x = U'b$  를 얻음. 이를  $d = U'b$  로 정의
3. 이제  $\Sigma V'x = d$  에서  $\Sigma$ 는 대각 행렬이므로 각 대각 성분에 대해 나누어  $V'x$ 를 구함
4. 마지막으로,  $V'x$ 에  $V$ 로 곱하여  $x$ 를 구합니다.



# SVD를 이용한 선형방정식 해법 예시

## 파이썬 코드 예시

- 이 예제는  $A$ 가 full-rank가 아니라서 유일한 해를 구할 수는 없음
- 대신에 최소 제곱 해를 구하는 방법을 보여줌

```
import numpy as np

#임의의 행렬 A와 벡터 b 정의
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
b = np.array([1, 0, 0])

#SVD 분해 수행
U, Sigma, VT = np.linalg.svd(A)

#Sigma를 대각 행렬로 변환하고 크기 맞추기
Sigma_mat = np.zeros_like(A, dtype=float)
Sigma_mat[:len(Sigma), :len(Sigma)] = np.diag(Sigma)

# $U^T * b$  계산
d = np.dot(U.T, b)

#Sigma의 의사 역행렬 계산
Sigma_pinv = np.linalg.pinv(Sigma_mat)

# $V^T x = Sigma_pinv * d$  계산
x_intermediate = np.dot(Sigma_pinv, d)

# $x = V * x_{\text{intermediate}}$  계산
x = np.dot(VT.T, x_intermediate)
```

# 반복 방법 (Iterative methods)

---

## 반복 방법을 이용한 선형방정식 해법

- 반복법은 큰 희소 행렬(sparse matrix)을 포함하는 선형 방정식을 해결하는 데 자주 사용됨
- 직접 해법은 유한한 수의 연산을 통해 정확한 해를 구하려 하지만, 반복법(iterative methods)은 일련의 근사치를 생성하여 점진적으로 정확한 해에 수렴

## 주요 특징

- **반복적 과정**: 반복법은 연속적인 근사치를 생성하고, 해가 정확한 해에 충분히 가까워질 때까지 계속됨. 이러한 방법들은 행렬-벡터 곱셈(matrix-vector products)을 포함하며 프로그래밍하기 쉬움
- **수렴 속도**: 반복법의 효율성은 수렴 속도에 따라 달라짐. 수렴이 항상 보장되지 않으며, 이는 행렬의 성질과 사용된 방법에 따라 달라짐
- **희소 행렬**: 큰 시스템에서 행렬의 대부분 요소가 0인 경우. 반복법은 이러한 희소성을 활용하여 비효율성을 줄여줌
- **정규화**: 반복법이 효과적이려면 방정식의 정규화가 필요할 수 있음. 이는 모든 대각 요소가 0이 아닌 상태로 행과 열을 재배열하는 것을 포함

# 야코비 방법(Jacobi Method)

---

## 선형방정식의 정규화

$$a_{11} x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3,$$

$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$$

$$\Rightarrow x_2 = (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$$

$$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}.$$

- 우변의  $x_i$ 는 미지수이지만,  $x^k$ 의 추정치를 이용해서 새로운 추정치  $x^{k+1}$ 으로 업데이트함

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11}$$

$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22}$$

$$x_3^{(k+1)} = (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33}.$$

## 알고리즘

---

### Algorithm 4 Jacobi iteration.

---

```
1: give initial solution  $x^{(0)} \in \mathbb{R}^n$ 
2: for  $k = 0, 1, 2, \dots$  until convergence do
3:   for  $i = 1 : n$  do
4:      $x_i^{(k+1)} = (b_i - \sum_{j \neq i} a_{ij}x_j^{(k)}) / a_{ii}$ 
5:   end for
6: end for
```

---

# 가우스-자이델 방법 (Gauss-Seidel Method)

- 좌변의 업데이트된 변수값을 바로 다음 변수의 업데이트에 사용함

$$x_1^{(k+1)} = \left( b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} \right) / a_{11}$$

$$x_2^{(k+1)} = \left( b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} \right) / a_{22}$$

$$x_3^{(k+1)} = \left( b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} \right) / a_{33}.$$

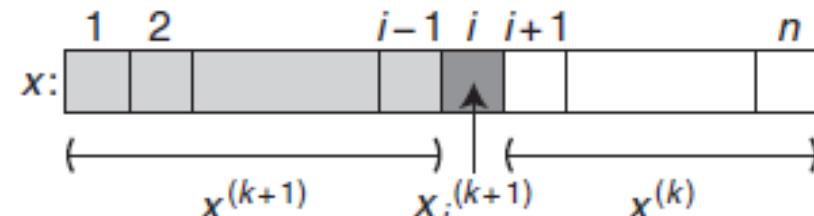
- 지난 값과 현재 값의 벡터를 저장할 필요가 없고, 과거 벡터값에 덮어 쓰기 가능함

---

## Algorithm 5 Gauss–Seidel iteration.

---

```
1: give initial solution  $x \in \mathbb{R}^n$ 
2: while not converged do
3:   for  $i = 1 : n$  do
4:      $x_i = \left( b_i - \sum_{j \neq i} a_{ij}x_j \right) / a_{ii}$ 
5:   end for
6: end while
```



# SOR (Successive overrelaxation)

---

## SOR 방법

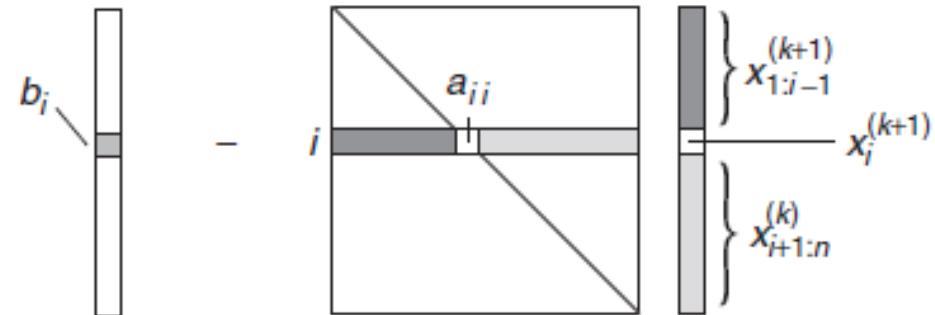
- Gauss-Seidel 방법의 수렴을 개선하기 위해서 다음의 업데이트 식을 이용

$$x_i^{(k+1)} = \omega x_{GS,i}^{(k+1)} + (1 - \omega)x_i^{(k)},$$

- $x_{GS,i}^{k+1}$ 는 Gauss-Seidel 방법으로 계산된  $x_i^{k+1}$  값
- $\omega$ 는 relaxation 파라미터
- 즉, SOR 방법은 Gauss-Seidel 방법의 값과 과거 SOR의 값의 선형 결합(linear combination)임

- Gauss-Seidel 방법 수식

$$x_{GS,i}^{(k+1)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right) / a_{ii}$$



- 위의 SOR 수식을 다시 쓰면

$$x_i^{(k+1)} = \omega \left( x_{GS,i}^{(k+1)} - x_i^{(k)} \right) + x_i^{(k)}.$$

- 이 식에서 괄호 안의  $x_{GS,i}^{k+1} - x_i^{(k)}$ 는 다음과 같이 표현 가능

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

- 이를 이용하면 SOR 수식은 다음과 같이 표현 가능

$$x_i^{(k+1)} = \omega x_{GS,i}^{(k+1)} + (1 - \omega)x_i^{(k)} \implies x_i = \omega(b_i - A_{i,1:n}x) / a_{ii} + x_i,$$

- $x_i^k$ 와  $x_i^{k+1}$ 는 같은 array에 저장할 수 있음

# SOR (Successive overrelaxation)

---

## SOR 방법

- 알고리즘

---

### Algorithm 6 SOR method.

---

```
1: give starting solution  $x \in \mathbb{R}^n$ 
2: while not converged do
3:   for  $i = 1 : n$  do
4:      $x_i = \omega(b_i - A_{i,1:n}x)/a_{ii} + x_i$ 
5:   end for
6: end while
```

---

## 완화 매개변수 $\omega$

- $\omega = 1$ 일 때, SOR은 Gauss-Seidel과 동일
- $\omega < 1$ 일 때, 수렴 속도 감쇠 → 발산하는 경우 수렴하도록 도움
- $\omega > 1$ 일 때, 수렴 속도 빨라짐

# 방정식 정규화와 수렴 속도

---

## 방정식의 정규화 선택은 수렴 속도에 영향을 미침

- 가우스-자이델(Gauss–Seidel)과 SOR 방법의 경우, 방정식이 반복되는 순서도 중요
- 수렴에 대한 공식적인 분석을 위해 계수 행렬  $A$ 를 다음과 같이 분해

$$A = L + D + U$$

- $L$  is lower triangular,  $D$  is diagonal, and  $U$  is upper triangular

## 반복 해법의 일반화

$$Mx^{k+1} = Nx^k + b$$

- **Jacobi:**  $M = D$ 이고,  $N = -(L + U)$
- **Gauss-Seidel:**  $M = D + L$ 이고,  $N = -U$
- **SOR:**  $Mx^{k+1} = Nx^k + \omega b$ 에서,  $M = D + \omega L$ 이고  $N = (1 - \omega)D - \omega U$

반복 방법은 행렬  $M^{-1}N$ 의 스펙트럴 반지름  $\rho$ 가 1보다 작을 때 수렴함

- SOR 방법의 경우,  $0 < \omega < 2$ 의 조건이 만족될 때만  $M^{-1}N$ 의 스펙트럴 반지름이 1보다 작아짐

# 수렴 조건

---

- 앞의 식에서  $A = M - N$  이므로,

$$Ax = b \quad \text{gives} \quad Mx = Nx + b$$

- $e^k = x^k - x$  라고 할 때,

$$Mx = N(x^k - e^k) + b$$

$$= Nx^k + b - Ne^k = Mx^{k+1} - Ne^k$$

- 이 식을 정리하면,

$$M(x^{k+1} - x) = Ne^k$$

$$e^{k+1} = M^{-1}Ne^k = (M^{-1}N)^k e^0$$

## 수렴조건

$$(M^{-1}N)^k \rightarrow 0 \quad \text{if and only if} \quad \rho(M^{-1}N) < 1$$

- 스펙트럴 반지름이 1보다 작은 조건에서 수렴함
- 이 조건은 선형 문제를 푸는 원래 문제보다 훨씬 더 많은 계산이 필요하기 때문에 실용적이지 않은 경우가 많으나, 선형 시스템을 반복적으로 풀어야 하는 상황에서는  $\omega$ 에 대한 최적의 값을 찾는 것을 권장
- Jacobi의 경우 대각선 우위(diagonal dominance)는 수렴을 위한 충분 조건임
- Gauss-Seidel의 경우 행렬 A가 symmetric positive-definite한 경우 반복이 항상 수렴함

# Spectral radius / Strictly diagonal dominant

---

## 스펙트럴 반지름 (Spectral radius)

- 행렬고유값(eigenvalues)의 절대값 중 최대값

$$\rho(A) = \max\{|\lambda_1|, |\lambda_2|, \dots, |\lambda_n|\}$$

## 절대대각우위(Strictly diagonal dominant)

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}| \quad i = 1, \dots, n$$

- 이 경우에  $A$ 의 스펙트럴 반지름이 다음을 만족함

$$\rho(M^{-1}N) \leq \|D^{-1}(L + U)\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1, j \neq i}^n \frac{|a_{ij}|}{|a_{ii}|} < 1.$$

# Stopping criterion

---

## 반복 방법의 중지 기준

- 오차가 충분히 작아지고 업데이트 벡터의 값이 크게 변하지 않을 조건
- Combination between absolute and relative error

$$\frac{|x_i^{(k+1)} - x_i^{(k)}|}{|x_i^{(k)}| + 1} < \epsilon \quad i = 1, 2, \dots, n,$$

- $\epsilon$ 은 tolerance 레벨임

## 최대반복회수 (max iteration)

**Algorithm 7** General structure of iterative methods.

---

```
1: initialize  $x^{(1)}$ ,  $x^{(0)}$ ,  $\epsilon$  and maxit
2: while not(converged( $x^{(0)}$ ,  $x^{(1)}$ ,  $\epsilon$ )) do
3:    $x^{(0)} = x^{(1)}$                                      # Store previous iteration in  $x^{(0)}$ 
4:   compute  $x^{(1)}$  with Jacobi, Gauss–Seidel, or SOR
5:   stop if number of iterations exceeds maxit
6: end while
```

---

# 블록 반복 방법

## Sparse systems of equations

- 특정 희소 방정식 시스템은 계수 행렬  $A$ 가 거의 블록 대각선 구조(대각선에는 상대적으로 조밀한 행렬이 있고 대각선 밖에는 매우 희박한 행렬)로 재배열할 수 있음
- 경제 분석에서는 여러 국가 모델이 상대적으로 적은 수의 무역 관계로 서로 연결되어 있는 국가 간 거시경제 모형이 이러한 경우
- 블록 반복 방법은 블록 대각선 분해로 정의된 하위 시스템을 반복하는 기법

선형 시스템  $Ax = b$ 에서  $A$ 가 대각 블록  $A_{ii}$ (정방행렬)로 구성된 경우

$$\begin{bmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \cdots & A_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}$$

or  $\sum_{j=1}^N A_{ij}x_j = b_i \quad i = 1, \dots, N$

---

### Algorithm 8 Block Jacobi method.

---

```
1: give initial solution  $x^{(0)} \in \mathbb{R}^n$ 
2: for  $k = 0, 1, 2, \dots$  until convergence do
3:   for  $i = 1 : N$  do
4:     solve  $A_{ii}x_i^{(k+1)} = b_i - \sum_{j=1, j \neq i}^N A_{ij}x_j^{(k)}$ 
5:   end for
6: end for
```

---

# 희소 선형 시스템 (Sparse linear systems)

## Sparse matrix

- 값이 0인 원소가 많아서 불필요한 연산이나 데이터 저장을 피하는 것이 이득인 행렬
- PDE를 이산화해서 해결하는 문제 등에서 적절한 처리 방법이 필요함

$$\begin{bmatrix} a_1 & b_1 & 0 & 0 & \cdots & 0 \\ c_1 & a_2 & b_2 & 0 & \cdots & 0 \\ 0 & c_2 & a_3 & b_3 & \cdots & 0 \\ 0 & 0 & c_3 & a_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_n \end{bmatrix}$$

## 삼중대각행렬(Tridiagonal system)

- banded matrix of bandwidth  $bw = 1$  ( $a_{ij} = 0$  if  $|i - j| > bw$ )
- LU분해를 이용해서 순차적으로 풀 수 있음

$$\underbrace{\begin{bmatrix} 1 & & & & \\ \ell_1 & 1 & & & \\ & \ell_2 & 1 & & \\ & & \ell_3 & 1 & \\ & & & \ell_4 & 1 \end{bmatrix}}_L \underbrace{\begin{bmatrix} u_1 & r_1 & & & \\ u_2 & r_2 & & & \\ u_3 & r_3 & & & \\ u_4 & r_4 & & & \\ u_5 & & & & \end{bmatrix}}_U = \underbrace{\begin{bmatrix} d_1 & q_1 & & & \\ p_1 & d_2 & q_2 & & \\ p_2 & d_3 & q_3 & & \\ p_3 & d_4 & q_4 & & \\ p_4 & d_5 & & & \end{bmatrix}}_A.$$

## 분해 알고리즘

Algorithm 9 Factorization of tridiagonal matrix.

```
1:  $u_1 = d_1$ 
2: for  $i = 2 : n$  do
3:    $\ell_{i-1} = p_{i-1}/u_{i-1}$ 
4:    $u_i = d_i - \ell_{i-1}q_{i-1}$ 
5: end for
```

# 최소자승법(Least square problem)

---

## 관찰값의 수가 매개변수의 수보다 많은 경우 선형방정식의 해

- "최적" 솔루션은 여러 가지 방법으로 결정될 수 있음
- 선형식  $Ax = b$ 에서  $b$ 는 관측값,  $A$ 는 독립변수,  $x$ 는 파라미터
- $A \in R^{m \times n}$ ,  $m > n$ 인 경우 overidentified 라고 하며 정확한 해를 구할 수 없음
- 잔차(residuals)

$$r(x) = Ax - b$$

- 최소자승법 (least square)

$$\min_{x \in R^n} g(x) = \frac{1}{2} r(x)' r(x) = \frac{1}{2} \sum_{i=1}^m r_i(x)^2,$$

## 잔차의 유클리디안 norm을 최소화

- 통계학에서 이러한 솔루션은 독립 동일 분포(i.i.d.) 잔차를 가지는 고전 선형 모델의 경우 최적의 선형불편추정치(BLUE)에 해당함
- 이 솔루션은 상대적으로 간단하고 효율적인 수치 절차로 얻을 수 있음
- 최소자승법 문제를 해결하기 위한 현대적인 수치 절차는 1960년대에 QR 분해와 SVD의 도입으로 발전하여 최근에는 큰 희소 행렬 시스템을 위한 방법이 개발 → 이 절차들은 선형 시스템의 해를 효율적으로 구하기 위해 중요한 도구임

# Normal equations

## Normal equations

- 1계 조건(first-order condition)

$$2A'Ax - 2A'b = 0$$

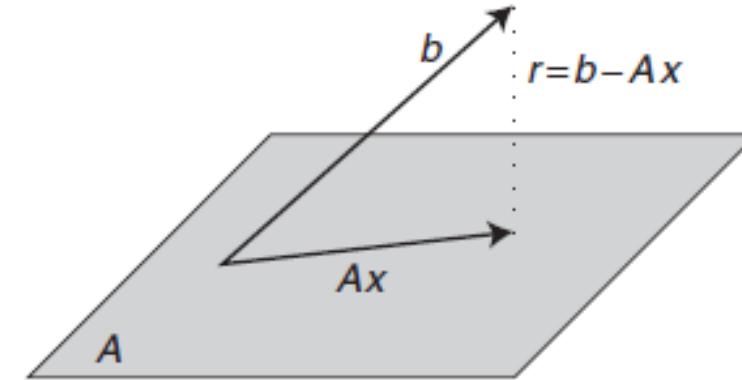
$$A'Ax = A'b \Rightarrow x = (A'A)^{-1}A'b$$

위의 식을 system of normal equations 라고 함

- 분산공분산 행렬(Variance-covariance matrix)

$$\sigma^2(A'A)^{-1}$$

- 이 식에서  $\sigma^2 = \frac{\|b-Ax\|_2^2}{m-n}$  으로 계산  $\Rightarrow (A'A)^{-1}$  및  $\|r(x)\|_2^2$  의 안정적인 계산 방법이 필요함



## 기하학적인 해석

- 벡터 b가 Ax 공간에 직교로 projection하는 값, 즉 가장 가까운 직선 거리의 값
- 직교조건:  $A'r = A'(b - Ax) = 0$ : 이 식으로 위의 normal equations 유도됨
- 만약 행렬 A가 full-rank인 경우, A'A는 positive-definite 하므로 콜레스키 분해 가능

# $(A'A)^{-1}$ 의 계산

## 직접 역행렬을 계산하지 않고 역행렬을 구하는 방법

- 콜레스키분해 이용  
 $A$ 가 full-rank일 때,  $A'A$ 는 양정부호이고,  
대칭행렬임

$$(A'A)^{-1} = (GG')^{-1} = G'^{-1}G^{-1}$$

- 삼각행렬  $G$ 의 역행렬 계산

$$T = G^{-1}$$

- 순차적으로 back-substitution 계산
- $T$ 의 원소  $t_{ij}$ 는 다음과 같이 계산 가능

$$t_{ij} = \begin{cases} 1/g_{ii} & i = j \\ -1/g_{ii} \sum_{k=j}^{i-1} g_{ik} t_{kj} & i \neq j \end{cases}$$

## 최소자승법 알고리즘

---

Algorithm 11 Least Squares with normal equations.

---

- 1: compute  $C = A'A$  # (only lower triangle needed)
  - 2: form  $\bar{C} = \begin{bmatrix} C & A'b \\ b'A & b'b \end{bmatrix}$
  - 3: compute  $\bar{G} = \begin{bmatrix} G & 0 \\ z' & \rho \end{bmatrix}$  # (Cholesky matrix)
  - 4: solve  $G'x = z$  # (upper triangular system)
  - 5:  $\sigma^2 = \rho^2 / (m - n)$
  - 6: compute  $T = G^{-1}$  # (inversion of triangular matrix)
  - 7:  $S = \sigma^2 T' T$  # (variance-covariance matrix)
-

# 잔차 제곱합 $\|r\|_2^2$ 의 계산

## 경계 행렬(bordered matrix) $\bar{A}$ 정의

$$\bar{A} = [A \ b]$$

- $\bar{C} = \bar{A}' \bar{A}$  라고 정의하고  $\bar{G}$ 를  $\bar{C}$ 의 콜레스키 분해 행렬이라고 할 때, 다음을 유도할 수 있음

$$\bar{C} = \begin{bmatrix} C & A'b \\ b'A & b'b \end{bmatrix} = \bar{G}\bar{G}' \quad \text{with} \quad \bar{G} = \begin{bmatrix} G & 0 \\ z' & \rho \end{bmatrix}.$$

$$\begin{bmatrix} C & A'b \\ b'A & b'b \end{bmatrix} = \begin{bmatrix} G & 0 \\ z' & \rho \end{bmatrix} \begin{bmatrix} G' & z \\ 0 & \rho \end{bmatrix} = \begin{bmatrix} GG' & Gz \\ z'G' & z'z + \rho^2 \end{bmatrix}$$

- where we recognize that  $G$  corresponds to the Cholesky matrix of  $A'A$  and

$$Gz = A'b \quad \text{and} \quad b'b = z'z + \rho^2$$

As  $r = b - Ax$  is orthogonal to  $Ax$ , we have

$$\|Ax\|_2^2 = (r + Ax)'Ax = b'Ax = \underbrace{b'AG^{-1}z}_{z'} = z'z$$

from which we derive

$$\begin{aligned} \|Ax - b\|_2^2 &= \underbrace{x'A'Ax}_{z'z} - 2\underbrace{b'Ax}_{z'z} + b'b \\ &= z'z - 2z'z + b'b = b'b - z'z = \rho^2. \end{aligned}$$

# QR분해를 이용한 최소자승법

## QR 분해

$$A = \begin{bmatrix} Q_1 & Q_2 \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} R_1 \\ \vdots \end{bmatrix}$$

- $\|Ax - b\|_2^2$ 에서  $A$ 행렬을  $QR$ 로 대체
- $Q'$ 을 곱해서 직교변환  $\|Q'QRx - Q'b\|_2^2$  - 잔차의 길이는 변하지 않음

$$\left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} Q'_1 \\ Q'_2 \end{bmatrix} b \right\|_2^2,$$

- 풀어서 다시 쓰면

$$\|R_1x - Q'_1b\|_2^2 + \|Q'_2b\|_2^2.$$

- 첫 번째항은  $R_1x = Q'_1b$ 에서 최소값  $\rightarrow x$ 의 해를 얻음
- 잔차제곱합은 두 번째 항  $\|Q'_2b\|_2^2$ 에 대응함
- $(A'A)^{-1} = (R'R)^{-1} = R^{-1}R'^{-1}$ 는 앞서 보았던 삼각행렬의 역행렬을 이용해서 계산함

# SVD를 이용한 최소자승법

## SVD의 적용

- 행렬  $A \in R^{m \times n}$  가 full rank가 아닌 경우에도 적용 가능함
- SVD를 통해  $A = U\Sigma V'$ 으로 분해하고, pseudoinverse(의사역행렬)  $A^+$ 를 정의

$$A^+ = V\Sigma^+U'$$

- $\Sigma^+$ 는 0이 아닌 특이값의 역수로 구성된 행렬
- 해  $x$ 는 다음과 같이 표현 가능 ( $r$ 은  $A$ 의 rank)

$$x = V \underbrace{\begin{bmatrix} \Sigma_r^{-1} & 0 \\ 0 & 0 \end{bmatrix}}_{A^+} U' b \quad \text{or} \quad x = \sum_{i=1}^r \frac{u_i'b}{\sigma_i} v_i,$$

## 잔차와 직교변환

- 직교변환 후 잔차 벡터와 유클리드 norm

$$z = V'x = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \quad \text{and} \quad c = U'b = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix},$$

$$\begin{aligned} \|b - Ax\|_2 &= \|U'(b - U\Sigma V'x)\|_2 = \|c - \Sigma z\|_2 \\ &= \left\| \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} - \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} c_1 - \Sigma_r z_1 \\ c_2 \end{bmatrix} \right\|_2 \end{aligned}$$

- $c_1 - \Sigma_r z_1 = 0$ 일 때 SSR 최소화

$$SSR = \|c_2\|_2^2 = \sum_{i=r+1}^m (u_i'b)^2$$

- $(A'A)^{-1} = V\Sigma_r^{-2}V'$ 로 계산 가능

## QR과 SVD를 이용한 최소자승법의 비교

---

- SVD 분해를 통한 해법은 최소자승 문제를 해결하기 위한 가장 비용이 많이 들지만 정확도가 가장 높은 수치적인 절차
- 이는 경제 금융과 관련된 문제에서는 거의 필요하지 않은 정확도임
- 현대 소프트웨어에서 최소자승 문제를 해결하기 위한 주요 방법은 QR 분해를 이용
- 그러나 만약  $m \gg n$ 이라면 정규 방정식 접근법은 약 절반의 연산과 메모리 공간만 필요

# 선형방정식의 해법과 역행렬 연산

---

## 선형방정식의 연산 프로세스

1. 정방행렬인지 확인, 정방행렬이 아니라면 QR 분해를 이용함
2. 정방행렬인 경우 하삼각행렬(lower triangular) 또는 상삼각행렬(upper triangular)인지 확인, 그렇다면 forward / back substitution 방법을 이용함
3. 대칭행렬(symmetric)이고 양정부호행렬(positive-definite)인지 확인, 그렇다면 촐레스키분해를 이용
4. 일반적인 정방행렬인 경우 LU분해를 이용해서 선형방정식의 해를 구함

## 역행렬 계산

- 가급적 역행렬을 구하는 연산은 하지 않는 것이 계산 효율적임 (선형방정식의 해를 구하는 것보다 2배 더 많은 계산 필요)
- 역행렬이 계산식에 포함되는 경우에 선형시스템의 해를 구해서 해결할 수 있는지 우선 확인
- 예를 들어,  $s = q' A^{-1} r$ 을 계산할 때 `s = q.T @ np.linalg.inv(A) @ r` 보다는
- `s = q.T @ np.linalg.solve(A, r)` 의 방법을 사용하는 것이 효율적임