밴드 행렬과 피벗팅

김기택
국민대학교 소프트웨어학과

밴드, 대칭 행렬

대칭 및 밴드 계수 행렬

- 많은 공학적 문제에서 행렬 요소가 희소한 (sparsely populated) 계수 행렬이 발생한다.
  - (희소 행렬: 행렬의 대부분의 요소가 0 인 계수 행렬) 이런 경우 일반적인 소거법이나 분해법을 사용하면 계산이 필요 없는 요소에 대해 계산 손실이 많이 발생한다. 계산을 할 필요가 있는 요소만을 저장하고 계산하면 계산 효율을 높일 수 있다.

  - 흔하게 나타나는 행렬의 형태는 삼중대각행렬(tridiagonal matrix) 이다.

$$\mathbf{A} = \begin{bmatrix} X & X & 0 & 0 & 0 \\ X & X & X & 0 & 0 \\ 0 & X & X & X & 0 \\ 0 & 0 & X & X & X \\ 0 & 0 & 0 & X & X \end{bmatrix}$$

대역폭(band width) = 3

  - 밴드 행렬이 $\mathbf{A} = \mathbf{LU}$ 형태로 분해되면, $\mathbf{L}$ 과 $\mathbf{U}$ 모두 $\mathbf{A}$ 의 밴드 구조를 유지한다.

$$\mathbf{L} = \begin{bmatrix} X & 0 & 0 & 0 & 0 \\ X & X & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 \\ 0 & 0 & X & X & 0 \\ 0 & 0 & 0 & X & X \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} X & X & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 \\ 0 & 0 & X & X & 0 \\ 0 & 0 & 0 & X & X \\ 0 & 0 & 0 & 0 & X \end{bmatrix}$$

삼중대각 계수행렬

- Doolittle 분해에 의한 **A x** = **b** 의 해를 고려하자. **A** 는 삼중대각 행렬이다.

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 & \cdots & 0 \\ c_1 & d_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_2 & d_3 & e_3 & \cdots & 0 \\ 0 & 0 & c_3 & d_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & d_n \end{bmatrix}$$

- **A** 의 0 이 아닌 원소를 벡터에 저장한다.

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{n-1} \end{bmatrix}$$

- 위와 같이 하면 저장 용량을 크게 절약할 수 있다.
  - 100 × 100 삼중 대각행렬은 99 + 100 + 99 = 298 위치에 저장 가능하다. (33:1 압축)

삼중 대각행렬에 LU분해법 적용 (1)

- c_{k-1} 계수를 없앰으로써 행 k 를 줄인다.

$$\text{열 } k \leftarrow \text{열 } k - (c_{k-1} / d_{k-1}) \times \text{row } (k-1), \quad k = 2, 3, \ldots, n$$

- d_k 의 변경사항은 다음과 같다. e_k 는 영향을 받지 않는다.

$$d_k \leftarrow d_k - (c_{k-1} / d_{k-1})\, e_{k-1}$$

- Doolittle 분해가 이루어지면, 이전에 c_{k-1} 이 차지하던 위치에 승수 $\lambda = c_{k-1} / d_{k-1}$ 을 저장

$$c_{k-1} \leftarrow c_{k-1} / d_{k-1}$$

- 분해 알고리즘은 다음과 같다.

```
for k in range(1, n):
    lam = c[k-1] / d[k-1]
    d[k] = d[k] – lam*e[k-1]
    c[k-1] = lam
```

삼중 대각행렬에 LU분해법 적용 (2)

- 다음으로 해를 구하는 단계를 살펴 본다. 우선 $\mathbf{Ly} = \mathbf{b}$ 단계를 보자.

$$\left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & \cdots & 0 & b_1 \\ c_1 & 1 & 0 & 0 & \cdots & 0 & b_2 \\ 0 & c_2 & 1 & 0 & \cdots & 0 & b_3 \\ 0 & 0 & c_3 & 1 & \cdots & 0 & b_4 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & 1 & b_n \end{array}\right]$$

- 분해과정에서 $\mathbf{c}$ 의 내용은 제거되고 승수로 대체된다. 다음은 $\mathbf{y}$ 에 대한 전진 대입 알고리즘이다.

```
y[0] = b[0]
for k in range(1, n):
    y[k] = b[k] – c[k-1] * y[k-1]
```

삼중 대각행렬에 LU분해법 적용 (3)

- **Ux** = **y** 를 나타내는 증강 계수 행렬은 다음과 같다.

$$[\mathbf{U} \mid \mathbf{y}] = \left[\begin{array}{cccccc|c} d_1 & e_1 & 0 & \cdots & 0 & 0 & y_1 \\ 0 & d_2 & e_2 & \cdots & 0 & 0 & y_2 \\ 0 & 0 & d_3 & \cdots & 0 & 0 & y_3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & d_{n-1} & e_{n-1} & y_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & d_n & b_n \end{array}\right]$$

  - **d** 의 내용이 분해단계에서 원래 값에서 변경되었음을 주목하라. (그러나, **e** 는 변경되지 않음)

- **x** 에 대한 해는 후진 대입에 의해 얻어진다. 알고리즘은 다음과 같다.

```
x[n-1] = y[n-1] / d[n-1]
for k in range(n-2, -1, -1):
    x[k] = (y[k] – e[k] * x[k+1]) / d[k]
```

LUdecomp3

- 이 모듈은 삼중 대각행렬 분해과정(LUdecomp3)과 전,후진 대입에 의한 해를 구하는 단계(LUsolve3)가 포함된다.

```
## module LUdecomp3
''' c,d,e = LUdecomp3(c,d,e).
    LU decomposition of tridiagonal matrix [c\d\e].
    Output {c},{d} and {e} are the diagonals of the
    decomposed matrix.
    x = LUsolve(c,d,e,b)
    Solves [c\d\e]{x} = {b}, where {c}, {d} and {e} are the
    vectors returned from LUdecomp3.
'''
def LUdecomp3(c,d,e):
    n = len(d)
    for k in range(1,n):
        lam = c[k-1]/d[k-1]
        d[k] = d[k] - lam*e[k-1]
        c[k-1] = lam
    return c,d,e
```

```
def LUsolve3(c,d,e,b):
    n = len(d)
    for k in range(1,n):
        b[k] = b[k] - c[k-1]*b[k-1]
        b[n-1] = b[n-1]/d[n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - e[k]*b[k+1])/d[k]
    return b
```

대칭 계수 행렬 (1)

- 종종 공학문제에서 계수 행렬은 대칭형과 밴드형이 있다. 따라서 이런 특수한 행렬에 대한 효율적인 알고리즘을 구성할 필요가 있다. 행렬 A 가 대칭이면 LU 분해는 다음과 같다.

$$\mathbf{A} = \mathbf{LU} = \mathbf{LDL}^T$$

  - **D** 는 대각선 행렬(대각선 요소가 모두 1인 행렬) 이다.
- Doolittle 분해를 하면 다음과 같아진다.

$$\mathbf{U} = \mathbf{DL}^T = \begin{bmatrix} D_1 & 0 & 0 & \cdots & 0 \\ 0 & D_2 & 0 & \cdots & 0 \\ 0 & 0 & D_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & D_n \end{bmatrix} \begin{bmatrix} 1 & L_{21} & L_{31} & \cdots & L_{n1} \\ 0 & 1 & L_{32} & \cdots & L_{n2} \\ 0 & 0 & 1 & \cdots & L_{n3} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

- 이를 전개하면 다음과 같다.

$$\mathbf{U} = \begin{bmatrix} D_1 & D_1L_{21} & D_1L_{31} & \cdots & D_1L_{n1} \\ 0 & D_2 & D_2L_{32} & \cdots & D_2L_{n2} \\ 0 & 0 & D_3 & \cdots & D_3L_{n3} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & D_n \end{bmatrix}$$

대칭 계수 행렬 (2)

- **D** 와 **L** 은 **U** 에서 쉽게 복구할 수 있으므로, 대칭 행렬의 분해 동안 **U** 만 저장한다는 것을 알 수 있다. 따라서 가우스 소거는 앞 식에 나타난 형태의 상삼각 행렬을 만들고, 대칭 행렬을 분해 하기에 충분하다.

- LU 분해 중에 사용할 수 있는 또 다른 저장 기법이 있는데, 다음 행렬을 살펴봄으로써 알 수 있다.

$$\mathbf{U}^* = \begin{bmatrix} D_1 & L_{21} & L_{31} & \cdots & L_{n1} \\ 0 & D_2 & L_{32} & \cdots & L_{n2} \\ 0 & 0 & D_3 & \cdots & L_{n3} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & D_n \end{bmatrix} \quad (2.25)$$

  - 여기서 **U** 는 $U_{ij} = D_i L_{ji}$ 로부터 회복할 수 있다.
  - 이 방법은 계산적으로 더 효율적인 해의 단계로 이어질 수 있다.

예제 2.9

가우스 소거의 결과로서 대칭 행렬 **A** 는 아래의 상삼각 행렬로 변형되었다. 원래의 행렬 **A** 를 결정하라.

$$\mathbf{U} = \begin{bmatrix} 4 & -2 & 1 & 0 \\ 0 & 3 & -3/2 & 1 \\ 0 & 0 & 3 & -3/2 \\ 0 & 0 & 0 & 35/12 \end{bmatrix}$$

[풀이] 먼저 **A** = **LU** 분해에서 **L** 을 찾는다. **U**의 각 행에 상응하는 대각선 요소로 나눈다.

$$\mathbf{L}^T = \begin{bmatrix} 1 & -1/2 & 1/4 & 0 \\ 0 & 1 & -1/2 & 1/3 \\ 0 & 0 & 1 & -1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

그러므로, **A** = **LU** 또는

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1/2 & 1 & 0 & 0 \\ 1/4 & -1/2 & 1 & 0 \\ 0 & 1/3 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 4 & -2 & 1 & 0 \\ 0 & 3 & -3/2 & 1 \\ 0 & 0 & 3 & -3/2 \\ 0 & 0 & 0 & 35/12 \end{bmatrix} = \begin{bmatrix} 4 & -2 & 1 & 0 \\ -2 & 4 & -2 & 1 \\ 1 & -2 & 4 & -2 \\ 0 & 1 & -2 & 4 \end{bmatrix}$$

예제 2.10

다음 대칭 행렬의 Doolittle 분해 $\mathbf{A} = \mathbf{LDL}^T$ 의 결과로 생기는 **L** 과 **D** 를 결정하라.

$$\mathbf{A} = \begin{bmatrix} 3 & -3 & 3 \\ -3 & 5 & 1 \\ 3 & 1 & 10 \end{bmatrix}$$

[풀이] 가우스 소거를 사용하여 **A** 의 상삼각 행렬 부분에 승수를 저장한다. 소거가 완료되면 행렬은 식 (2.25)의 $\mathbf{U}^*$ 의 형태를 갖게 된다.
첫번째 단계에서 제거할 항목은 기본 작업을 사용하는 A_{21} 과 A_{31} 이다.

$$\text{열 } 2 \leftarrow \text{열 } 2 - (-1) \times \text{열 } 1$$
$$\text{열 } 3 \leftarrow \text{열 } 3 - (1) \times \text{열 } 1$$

A_{12} 와 A_{13} 이 차지하는 위치에 승수 (-1 과 1)을 저장하면,

$$\mathbf{A}' = \begin{bmatrix} 3 & -1 & 1 \\ 0 & 2 & 4 \\ 0 & 4 & 7 \end{bmatrix}$$

두번째 단계는 다음과 같다.

예제 2.10 (continued)

$$\text{열 } 3 \leftarrow \text{열 } 3 - 2 \times \text{열 } 2$$

A_{23} 을 승수 2로 덮어 쓰면

$$\mathbf{A}'' = [\mathbf{0} \setminus \mathbf{D} \setminus \mathbf{L}^T] = \begin{bmatrix} 3 & -1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & -1 \end{bmatrix}$$

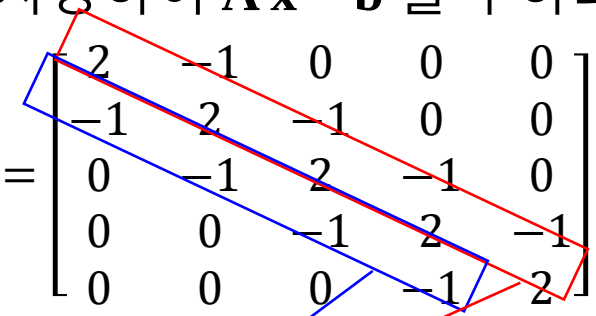
그러므로,

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

예제 2.11

LUcomp3 및 LUsolve3 함수를 사용하여 $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ 를 구하라.

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 5 \\ -5 \\ 4 \\ -5 \\ 5 \end{bmatrix}$$



```
## example2_11
import numpy as np
from LUdecomp3 import *

d = np.ones((5))*2.0
c = np.ones((4))*(-1.0)
b = np.array([5.0, -5.0, 4.0, -5.0, 5.0])
e = c.copy()
c,d,e = LUdecomp3(c,d,e)
x = LUsolve3(c,d,e,b)
print("\nx =\n",x)
```

피벗팅

방정식의 순서와 해의 오차 (1)

- 방정식의 순서에 따라 결과에 큰 영향을 준다. 다음 방정식을 고려하여 보자.

$$2x_1 - x_2 = 1$$
$$-x_1 + 2x_2 - x_3 = 0$$
$$-x_2 + x_3 = 0$$

- 대응하는 증강 계수 행렬은 다음과 같다.

$$[\mathbf{A}|\mathbf{b}] = \begin{bmatrix} 2 & -1 & 0 & 1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix} \quad (a)$$

- 위의 식은 가우스 소거를 통해 해를 얻는데 문제가 없다는 의미에서 "올바른 순서"에 있다. 이제 첫번째 식과 세번째 식을 교환하여 보자.

$$[\mathbf{A}|\mathbf{b}] = \begin{bmatrix} 0 & -1 & 1 & 0 \\ -1 & 2 & -1 & 0 \\ 2 & -1 & 0 & 1 \end{bmatrix} \quad (b)$$

- 여전히 해는 동일하여야 한다. 그러나 첫번째 소거 단계에서 피벗 요소가 0 이므로 문제가 발생한다.

방정식의 순서와 해의 오차 (2)

- 이 예는 동일한 방정식이라고 해도 순서에 따라 해를 구하는 방법에 영향이 크다는 것을 알 수 있다. 피벗 요소가 0 이 아니라고 해도 다른 요소와 비교할 때 매우 작은 경우를 살펴보자.

$$[\mathbf{A}|\mathbf{b}] = \begin{bmatrix} \varepsilon & -1 & 1 & 0 \\ -1 & 2 & -1 & 0 \\ 2 & -1 & 0 & 1 \end{bmatrix}$$

- 만약 ε 가 0 으로 근접한다면 $(\varepsilon \to 0)$ 식 (a) 와 (b)의 해는 동일하여야 한다. 가우스 소거를 거친 계수 행렬은

$$[\mathbf{A}'|\mathbf{b}'] = \begin{bmatrix} \varepsilon & -1 & 1 & 0 \\ 0 & 2 - 1/\varepsilon & -1 + 1/\varepsilon & 0 \\ 0 & -1 + 2/\varepsilon & -2/\varepsilon & 1 \end{bmatrix}$$

- 컴퓨터에서는 모든 숫자가 유효숫자로 반올림된다. ε 이 매우 작으면 $1/\varepsilon$ 는 매우 크고 $2 - 1/\varepsilon$ 와 같은 요소는 $-1/\varepsilon$ 로 반올림된다. 따라서 충분히 작은 ε 에 대해서 실제 식은 다음과 같아진다.

방정식의 순서와 해의 오차 (3)

$$[\mathbf{A}'|\mathbf{b}'] = \begin{bmatrix} \varepsilon & -1 & 1 & 0 \\ \mathbf{0} & \mathbf{-1/\varepsilon} & \mathbf{1/\varepsilon} & \mathbf{0} \\ \mathbf{0} & \mathbf{2/\varepsilon} & \mathbf{-2/\varepsilon} & \mathbf{1} \end{bmatrix}$$

- 위 식을 살펴 보면 두번째와 세번째 식이 서로 모순을 일으킨다. (두 식은 동일한 식이지만 외부조건 상수가 다르다.)
  - 그러나, 이 문제는 소거 전에 첫번째 행과 두번째 혹은 세번째 행이 서로 교환되면 발생하지 않을 수 있다.
  - 이 문제는 다른 요소에 비해 무척 작은 요소로 인하여 발생하는 잠재적인 문제점을 보여준다.

- 이러한 문제는 **피벗팅을 통해 피할 수 있다**.

대각 지배

- 각 대각 요소가 같은 행에 있는 다른 요소의 합보다 큰 경우 $n \times n$ 행렬 $\mathbf{A}$ 가 대각선으로 지배적이라고 한다. (절대값 기준으로 판단) **대각 지배**는,

$$|A_{ij}| > \sum_{j=1, j \neq i}^{n} |A_{ij}| \quad (i = 1, 2, \ldots, n)$$

- 예를 들어 다음 행렬은 대각 지배가 아니다.

$$\begin{bmatrix} -2 & 4 & -1 \\ 1 & -1 & 3 \\ 4 & -2 & 1 \end{bmatrix}$$

- 그러나 행을 재정렬하면 대각지배행렬이 된다.

$$\begin{bmatrix} 4 & -2 & 1 \\ -2 & 4 & -1 \\ 1 & -1 & 3 \end{bmatrix}$$

- 계수 행렬이 대각지배행렬이면 피벗팅의 이점이 없다. 이미 최적의 순서로 정렬되어 있기 때문이다. **피벗팅은 식을 재정렬하여 대각지배에 가깝도록 하는 것**이다.

스케일 된 행 피벗을 사용한 가우스 소거 (1)

- 행 피벗의 가우스 소거에 의한 A x = b 의 해를 생각해 보자.
  - 피벗팅은 계수 행렬의 대각 지배를 개선하는 것을 목표로 한다. (피벗행의 피벗 요소를 다른 요소에 비교하여 가능한 크게 만드는 것)
- 다음과 같은 요소가 있는 배열 **s** 를 만들면 비교가 쉽다.

$$s_i = \max_j |A_{ij}|, \qquad i = 1, 2, \ldots, n$$

  - s_i 는 행 i 의 축척비율(scale factor)라고 하며, **A** 의 i 번째 행에서 가장 큰 요소의 절대값을 포함한다. 벡터 **s** 는 다음 알고리즘으로 얻을 수 있다.

```
for i in range(n):
    s[i] = max(abs(a[i, : ]))
```

- 요소 A_{ij} 의 상대 크기 (i 번째 행에서 가장 큰 요소에 대한 비율)는 다음과 같다.

$$r_{ij} = \frac{|A_{ij}|}{s_i}$$

스케일 된 행 피벗을 사용한 가우스 소거 (2)

- 소거 단계에서 k 번째 행이 피벗행이 된 단계를 가정하자. 계수 행렬은 다음과 같다.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1k} & \cdots & A_{1j} & \cdots & A_{1n} & b_1 \\ 0 & A_{22} & A_{23} & \cdots & A_{2k} & \cdots & A_{2j} & \cdots & A_{2n} & b_2 \\ 0 & 0 & A_{33} & \cdots & A_{3k} & \cdots & A_{3j} & \cdots & A_{3n} & b_3 \\ \vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{kk} & \cdots & A_{kj} & \cdots & A_{kn} & b_k \\ \vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{ik} & \cdots & A_{ij} & \cdots & A_{in} & b_i \\ \vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{nk} & \cdots & A_{nj} & \cdots & A_{nn} & b_n \end{bmatrix}$$

- A_{kk} 를 자동적으로 피벗요소로 받아들이지 않고 A_{kk} 아래의 k 번째 열에서 "더 나은" 피벗을 찾는다. 즉, 가장 큰 상대 크기를 가지는 A_{kk} 요소를 선택한다.
  - 다음과 같은 p 를 선택하여 k 와 p 행을 교환하고 소거 과정을 진행한다. 대응하는 행교환은 스케일 인자 배열 **s** 에서도 수행되어야 한다.

$$r_{pk} = \max_j(r_{jk}), \qquad j \geq k$$

피벗팅 알고리즘

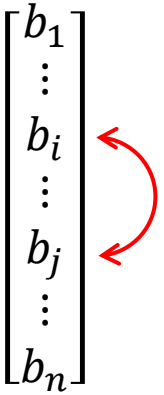
- 피벗팅을 수행하는 알고리즘은 다음과 같다.

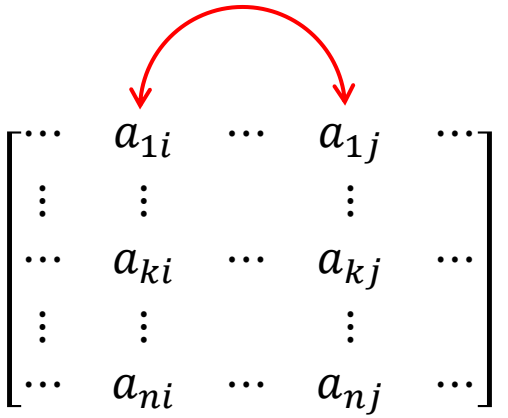
```
for k in range(0,n-1):
    # Find row containing element with largest relative size
    p = argmax(abs(a[k:n,k])/s[k:n]) + k     상대크기가 가장 큰 요소의 위치를 찾음

    # If this element is very small, matrix is singular
    if abs(a[p,k]) < tol: error.err('Matrix is singular')     일정 조건보다 작은 피벗 요소가 있으면 특이행렬로 판단

    # Check whether rows k and p must be interchanged
    if p != k:                                     찾아진 요소가 현재 요소와 같지 않으면 행 교환 실시
        # Interchange rows if needed
        swap.swapRows(b,k,p)    외부 조건 벡터 교환
        swap.swapRows(s,k,p)    축척 비율 벡터 교환
        swap.swapRows(a,k,p)    계수 행렬 교환
    # Proceed with elimination
```

행 교환 알고리즘

- swapRows 는 행렬 또는 벡터 v 의 행 i 와 j 를 교환하며, swapCols 는 행렬의 열 i 와 j 를 교환한다.

```
## module swap
''' swapRows(v,i,j).                                    모듈 내 함수 설명
    Swaps rows i and j of a vector or matrix [v].
    swapCols(v,i,j).
    Swaps columns of matrix [v].
'''
def swapRows(v,i,j):        벡터와 행렬의 행 교환 함수
    if len(v.shape) == 1:   v 가 벡터인 경우
        v[i],v[j] = v[j],v[i]
    else:
        v[[i,j],:] = v[[j,i],:]   v 가 행렬인 경우

def swapCols(v,i,j):        행렬의 열 교환 함수
    v[:,[i,j]] = v[:,[j,i]]
```





gaussPivot 프로그램

- 피벗팅 부분을 제외하면 gaussElimin 과 동일하다.

```
## module gaussPivot
''' x = gaussPivot(a,b,tol=1.0e-12)
    Solves [a]{x} = {b} by Gauss elimination with
    scaled row pivoting
'''
import numpy as np
import swap, error
def gaussPivot(a,b,tol=1.0e-12):
    n = len(b)
    # Set up scale factors
    s = np.zeros(n)
    for i in range(n):
        s[i] = max(np.abs(a[i,:]))
    for k in range(0,n-1):
        # Row interchange, if needed
        p = np.argmax(np.abs(a[k:n,k])/s[k:n]) + k
        if abs(a[p,k]) < tol: error.err('Matrix is singular')
        if p != k:
            swap.swapRows(b,k,p)
            swap.swapRows(s,k,p)
            swap.swapRows(a,k,p)
    for i in range(k+1,n):
        if a[i,k] != 0.0:
            lam = a[i,k]/a[k,k]
            a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
            b[i] = b[i] - lam*b[k]
    if abs(a[n-1,n-1]) < tol: error.err('Matrix is singular')
    b[n-1] = b[n-1]/a[n-1,n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - np.dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

특이행렬 판정 한계

각 행의 축척 비율 벡터 계산

피벗팅 단계

외부 조건 벡터 교환

축척 비율 벡터 교환

계수 행렬 교환

소거 단계 진행

 # Elimination

후진 대입 단계

 # Back substitution

LUpivot 프로그램

- LU 분해법에서도 피벗팅은 프로그램의 안정성을 높여준다.
  - 분해 단계에서 행 교환을 기록함으로써 이를 배열 순서 벡터에 기록한다.
  - 전진, 후진 대입 단계를 진행하기 전 배열 순서 벡터의 순서 대로 상수 벡터의 요소를 재정렬한다.

```
def LUdecomp(a,tol=1.0e-9):
    n = len(a)
    seq = np.array(range(n))  배열 순서 벡터
    # Set up scale factors
    s = np.zeros((n))
    for i in range(n):
        s[i] = max(abs(a[i,:]))
    for k in range(0,n-1):
    # Row interchange, if needed
        p = np.argmax(np.abs(a[k:n,k])/s[k:n]) + k
        if abs(a[p,k]) < tol: error.err('Matrix is singular')
        if p != k:
            swap.swapRows(s,k,p)
            swap.swapRows(a,k,p)
            swap.swapRows(seq,k,p)
    # Elimination
for i in range(k+1,n):    소거 단계
        if a[i,k] != 0.0:
            lam = a[i,k]/a[k,k]
            a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
            a[i,k] = lam
    return a,seq

def LUsolve(a,b,seq):    전진, 후진 대입 단계
    n = len(a)
    # Rearrange constant vector; store it in [x]
    x = b.copy()
    for i in range(n):    행 교환 순서에 맞춰 외부 조건 벡터 순서 재정렬
        x[i] = b[seq[i]]
    # Solution
    for k in range(1,n):
        x[k] = x[k] - np.dot(a[k,0:k],x[0:k])
    x[n-1] = x[n-1]/a[n-1,n-1]
    for k in range(n-2,-1,-1):
        x[k] = (x[k] - np.dot(a[k,k+1:n],x[k+1:n]))/a[k,k]
    return x
```

피벗팅 시기

- 피벗팅에는 두 가지 단점이 있다.
  - 계산 비용이 증가한다.
  - 대칭의 파괴와 계수 행렬의 밴드 구조 형성이다

- 특히 후자는 계수 행렬이 대칭이고 밴드 일 때 문제가 된다.
  - 대부분의 공학 문제에서 발생하는 계수 행렬은 대각 지배적이므로 피벗이 큰 도움을 주지 못한다.
  - 계수 행렬이 밴드 일 경우 피벗팅은 비생산적일 수 있다.
  - 양의 정부호 행렬(양정, positive definite)과 더 낮은 차수의 대칭 행렬은 피벗팅 혜택이 거의 없다.

- 피벗팅이 반올림 오류를 제어하는 유일한 수단이 아니라는 점을 인식하여야 한다.
  - 교과서 예제 2.12는 이러한 규칙을 따르지 않는 예 이다. (각자 공부하기 바란다.)