13.1 피벗화 (Pivoting)

1. 피벗 요소가 0에 가까울 때 발생하는 문제

- 가우스 소거법에서 피벗 요소가 0이거나 0에 매우 가까우면 계산 과정에서 큰 오차가 발생할 수 있습니다. 이러한 경우의 문제를 해결하는 것이 피벗화입니다.

2. 부분 피벗화 (Partial Pivoting)

- 행렬에서 각 행이 정규화되기 전에 피벗 요소를 선택하는데, 피벗 요소로 선택될 값이 0에 가까운 경우 오차가 커질 수 있습니다.
- 부분 피벗화는 가장 큰 계수를 가진 행으로 피벗 행을 교환함으로써 오차를 줄이는 방법입니다.
- 피벗화는 변수들의 순서나 열을 바꾸지 않고 행끼리만 교환합니다.

3. 완전 피벗화 (Complete Pivoting)

- 부분 피벗화와는 달리, 완전 피벗화는 행뿐만 아니라 열도 교환하여 가장 큰 요소를 선택할 수 있도록 합니다.
- 완전 피벗화는 계산적으로 더 복잡하고 변수가 순서대로 나오지 않을 수 있어 잘 사용되지 않습니다.

예시 문제 풀이

주어진 문제는 다음과 같습니다:

$$0.0003x_1 + 3.0000x_2 = 2.0001$$
$$1.0000x_1 + 1.0000x_2 = 1.0000$$

1단계: 가우스 소거법 시작

주어진 방정식을 행렬 형태로 변환합니다:

$$\begin{bmatrix} 0.0003 & 3.0000 & 2.0001 \\ 1.0000 & 1.0000 & 1.0000 \end{bmatrix}$$

2단계: 첫 번째 행 정리

첫 번째 피벗 요소를 기준으로 나누고, 행렬을 정리합니다. 첫 번째 행의 피벗 요소를 1로 만들기 위해 0.0003으로 나눕니다.

$$\begin{bmatrix} 1.0000 & 10000 & 6667 \\ 1.0000 & 1.0000 & 1.0000 \end{bmatrix}$$

3단계: 두 번째 행에서 첫 번째 요소 제거

두 번째 행에서 첫 번째 요소를 없애기 위해 첫 번째 행을 빼줍니다:

$$\begin{bmatrix} 1.0000 & 10000 & 6667 \\ 0 & -9999 & -6666 \end{bmatrix}$$

4단계: 후진 대입 (Back Substitution)

두 번째 행에서 x_2를 구할 수 있습니다:

$$x_2 = \frac{2}{3}$$

구한 x2x_2x2 값을 첫 번째 식에 대입하여 x1x_1x1을 구합니다:

$$x_1 = \frac{2.0001 - 3 \cdot \frac{2}{3}}{0.0003} = 0.0003$$

최종적으로, $x_1 = 0.0003, x_2 = \frac{2}{3}$임을 알 수 있습니다.

첫 번째 계산 결과

첫 번째로 가우스 소거법을 적용한 결과는 다음과 같습니다:

$$x_2 = \frac{2}{3}, \quad x_1 = \frac{2.0001 - 3 \times \frac{2}{3}}{0.0003} = \frac{1}{1} - \frac{2}{3} = 1 - \frac{2}{3} = \frac{1}{3}$$

이 계산 결과는 **소수점 단위의 작은 차이**가 변수들의 값에 **큰 영향**을 미친다는 것을 보여줍니다. 따라서, **유효 숫자**의 개수에 따라 계산 결과가 달라질 수 있습니다.

유효 숫자와 민감도 분석

표에서는 **유효 숫자**가 달라질 때, 즉 소수점 이하의 자릿수에 따라 계산된 값들이 어떻게 달라지는지 보여줍니다.

- **유효 숫자 3**: $x_1 = -3.33$, 상대 오차: 1099
- **유효 숫자 4**: $x_1 = 0.0000$, 상대 오차: 0.01
- **유효 숫자 5**: $x_1 = 0.0000$, 상대 오차: 10
- **유효 숫자 6**: $x_1 = 0.3300$, 상대 오차: 0.1
- **유효 숫자 7**: $x_1 = 0.333333$, 상대 오차: 0.00001

이 표를 통해 확인할 수 있는 것은, **소수점 자릿수가 적을수록 계산 결과의 오차가 크다**는 점입니다. 즉, 높은 정확도를 위해서는 더 많은 유효 숫자를 사용해야 함을 보여줍니다.**가우스 소거법 + 피벗화 적용**

피벗화를 적용하여 계산을 개선한 과정은 다음과 같습니다.

1. 행렬에 피벗화를 적용하여 x_2와 x_1을 계산합니다:

$$\begin{bmatrix} 1.0000 & 1.0000 & 1.0000 \\ 0 & 2.9997 & 1.9998 \end{bmatrix}$$

1. 피벗화 후, $x_2 = \frac{2}{3}$, $x_1 = 1 - \frac{2}{3} = \frac{1}{3}$이라는 값을 얻게 됩니다.
2. 피벗화를 적용한 후에는 오차가 크게 줄어들며, 더 정확한 값을 얻을 수 있습니다.

유효 숫자와 피벗화 후 민감도 분석

피벗화를 적용한 후의 유효 숫자에 따른 계산 결과는 더 정확해집니다:

- 유효 숫자 3: $x_1 = 0.333$, 상대 오차: 0.1
- 유효 숫자 4: $x_1 = 0.3333$, 상대 오차: 0.01
- 유효 숫자 5: $x_1 = 0.33333$, 상대 오차: 0.001
- 유효 숫자 6: $x_1 = 0.333333$, 상대 오차: 0.0001
- 유효 숫자 7: $x_1 = 0.3333333$, 상대 오차: 0.00001

13.2 Gauss 소거법 개선을 위한 알고리즘축척화(Scaling)

축척화는 행렬의 크기를 표준화하여 반올림 오차를 최소화하는 기법입니다.

- 각 행렬의 요소들을 비슷한 크기로 맞추어 계산할 때 발생할 수 있는 오차를 줄입니다.

예제 문제 풀이

(a) 가우스 소거법 + 피벗화 적용

다음의 연립 방정식을 가우스 소거법과 피벗화를 사용해 풉니다:

$$2x_1 + 100,000x_2 = 100,000$$
$$x_1 + x_2 = 2$$

이를 행렬 형태로 변환하면:

$$\begin{bmatrix} 2 & 100000 & 100000 \\ 1 & 1 & 2 \end{bmatrix}$$

1. 첫 번째 피벗을 선택하고, 두 번째 행에서 첫 번째 요소를 제거합니다:

$$\begin{bmatrix} 1 & 50000 & 50000 \\ 1 & 1 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 50000 & 50000 \\ 0 & -49999 & -49998 \end{bmatrix}$$

1. 후진 대입을 통해 값을 계산합니다:

$$x_2 = 1.00 \quad x_1 = 0$$

하지만 이 과정에서 반올림 오차가 발생하여, x_1에서 100% 오차가 발생합니다. 이는 원래 정확한 값이 $x_1 = 1.00002, x_2 = 0.99998$임을 알 수 있습니다. 그러나 유효숫자를 3개로 반올림하면 $x_1 = x_2 = 1.00$로 간주됩니다.

(b) 축척화 + 피벗화 적용

축척화를 먼저 적용한 후 피벗화를 사용하면 다음과 같이 개선됩니다:

1. 행렬을 축척화하여 크기를 맞춥니다:

$$\begin{bmatrix} 2 & 10^5 & 10^5 \\ 1 & 1 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 5\times10^4 & 5\times10^4 \\ 1 & 1 & 2 \end{bmatrix}$$

1. 피벗화를 적용하여 계산하면:

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

최종적으로 $x_2 = 1$, $x_1 = 0$을 얻게 됩니다.

(c) 축척화 → 피벗화 → 원래 계수 사용

1. 축척화 적용

원래 행렬을 축척화하여 반올림 오차를 줄입니다. 행렬의 각 값을 일정한 크기로 축소하여 계산이 더 정확하게 이루어지도록 합니다.

$$\begin{bmatrix} 1 & 1 & 2 \\ 0.00002 & 0.00002 & 0.00004 \end{bmatrix}$$

행렬을 축척화한 후에, 피벗화를 적용하면 더 정확한 계산 결과를 얻을 수 있습니다.

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

이 결과로부터 $x_2 = 1.00, x_1 = 1.00$을 얻게 됩니다.

2. 피벗화와 원래 계수 사용

축척화 후 피벗화만을 사용하여 계산할 수도 있습니다. 여기서, 축척화된 행렬을 피벗화를 통해 변환한 후 원래 계수를 사용해 연립 방정식을 풉니다.

$$\begin{bmatrix} 1 & 1 & 2 \\ 2 & 100000 & 100000 \end{bmatrix}$$

이를 통해 $x_2 = 1.00$, $x_1 = 1.00$이라는 값을 얻을 수 있습니다. 유효 숫자 3개로 제한된 상태에서 이 값이 나온다는 점도 확인할 수 있습니다.

축척화의 이점 및 한계

- **이점**: 축척화는 반올림 오차를 최소화하기 위해 매우 유용하며, 숫자의 스케일 차이로 인한 오차를 줄여줍니다.
- **한계**: 축척화는 피벗화와 함께 사용되지 않을 경우 여전히 반올림 오차가 발생할 수 있습니다.

결론

- 피벗화 기준을 정할 때 축척화를 사용하여 더 나은 결과를 얻을 수 있지만, 계산 과정에서 축척화된 값을 기준으로 하면 여전히 오차가 발생할 수 있습니다.
- 피벗화를 적용한 후 원래 계수로 복원하여 계산하는 것이 더 정확한 방법입니다.

Gauss 소거법 알고리즘 개선점

1. **피벗화 판단을 위한 축척화**

- 연립 방정식을 축척화하여 피벗화할 때 반올림 오차를 줄이는 기준을 제공합니다.
- 축척화의 과정에서 발생하는 반올림 오차는 피벗화를 통해 다시 개선될 수 있습니다.

1. **상부 삼각 행렬 형성 중 대각선 원소가 0이 되는 문제**

- 피벗화를 사용하지 않으면 대각선 원소가 0이 되는 문제가 발생할 수 있으며, 이는 축척화로만 해결할 수 없습니다.
- 따라서, 축척화와 피벗화를 함께 사용하는 것이 중요합니다.

1. **근접일 때도 오차 발생**

- 반올림 오차가 근접한 상황에서 오차가 증폭될 수 있기 때문에, 축척화와 피벗화가 필수적으로 필요합니다.

13.3 Gauss-Jordan법

Gauss-Jordan법은 전통적인 가우스 소거법에서 한 단계 더 나아간 방식입니다. 가우스 소거법이 상부 삼각 행렬까지만 변환한다면, Gauss-Jordan법은 추가적인 단계를 통해 **단위 행렬**을 만들어 직접 해를 구합니다. 이를 통해 행렬을 더 간단하게 해석할 수 있습니다.

Gauss-Jordan법의 특징:

- **전진 소거 단계**에서 피벗 요소를 사용하여 모든 방정식의 미지수를 제거합니다.
- **후진 소거** 대신, 모든 행을 피벗 요소로 정규화합니다. 즉, 행렬의 대각선에 있는 값이 모두 1이 되도록 변환합니다.

이 방식은 행렬을 **완전한 단위 행렬**로 변환하여 결과적으로 해를 더 쉽게 구할 수 있습니다.

절차 설명:예제 문제:

주어진 방정식:

$$
\begin{aligned}
3x_1 - 0.1x_2 - 0.2x_3 &= 7.85 \\
0.1x_1 + 7x_2 - 0.3x_3 &= -19.3 \\
0.3x_1 - 0.2x_2 + 10x_3 &= 71.4
\end{aligned}
$$

위의 연립 방정식을 Gauss-Jordan법을 통해 풉니다. 이 방정식을 행렬로 변환하면:

$$
\begin{bmatrix}
3 & -0.1 & -0.2 & 7.85 \\
0.1 & 7 & -0.3 & -19.3 \\
0.3 & -0.2 & 10 & 71.4
\end{bmatrix}
$$

1단계: 피벗 요소를 사용한 소거

첫 번째 열에서 피벗 요소를 사용해 첫 번째 행을 정규화합니다:

$$\begin{bmatrix} 1 & -0.0333333 & -0.0666667 & 2.616667 \\ 0.1 & 7 & -0.3 & -19.3 \\ 0.3 & -0.2 & 10 & 71.4 \end{bmatrix}$$

- 첫 번째 행의 피벗 요소(3)를 1로 만들어 정규화하고, 다른 행을 이 첫 번째 행을 기준으로 수정합니다.

2단계: 나머지 행들 소거

다음으로 두 번째 열에서 피벗 요소(7)를 사용해 두 번째 행을 정규화하고, 다른 행들을 소거합니다:

$$\begin{bmatrix} 1 & -0.0333333 & -0.0666667 & 2.616667 \\ 0 & 7.003333 & -0.2933333 & -19.56167 \\ 0.3 & -0.2 & 10 & 71.4 \end{bmatrix}$$

3단계: 세 번째 행 소거

마지막으로 세 번째 행의 피벗 요소(10)를 1로 만들어 소거합니다:

$$\begin{bmatrix} 1 & -0.0333333 & -0.0666667 & 2.616667 \\ 0 & 1 & -0.04184848 & -2.79320 \\ 0 & 0 & 10.0200 & 70.6150 \end{bmatrix}$$

이제 세 번째 행까지 완벽하게 소거한 후, 마지막으로 세 번째 행에서 피벗을 적용하면 다음과 같은 행렬이 나옵니다:

$$\begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & x_2 \\ 0 & 0 & 1 & x_3 \end{bmatrix}$$

이 행렬은 **단위 행렬**의 형태이며, 이를 통해 각 미지수의 값을 직접 얻을 수 있습니다:

$$x_1 = 3, \quad x_2 = -2, \quad x_3 = 7$$

Gauss-Jordan법과 Gauss 소거법의 차이점:

1. **소거 방식**:

- Gauss 소거법: 상부 삼각 행렬까지만 변환하고 후진 대입을 통해 해를 구합니다.
- Gauss-Jordan법: 모든 열에 대해 피벗화를 적용하여 단위 행렬을 형성합니다. 후진 대입이 필요 없고, 해를 바로 얻을 수 있습니다.

1. **단위 행렬**:

- Gauss 소거법은 상부 삼각 행렬을 만들지만, Gauss-Jordan법은 모든 미지수를 1로 만들어 완전한 단위 행렬로 변환합니다.

1. **해석적 간편함**:

- Gauss-Jordan법은 마지막에 단위 행렬이 되기 때문에 결과를 바로 얻을 수 있어, 직관적으로 해를 확인할 수 있습니다.

```
function NA13_2_GaussEliminationImproved()
    G = [ 3, -0.1, -0.2, 7.85;
          0.3, -0.2, 10, 71.4;
          0.1, 7, -0.3, -19.3];
    N = 3;

    %% 전진 소거
    for i = 1:N-1
        G = pivot(G, i, N); % pivoting
        for j = i+1:N
            G(j,:) = G(j,:) - G(j,i) / G(i,i) * G(i,:);
        end
    end

    %% 후진 대입
    X = zeros(N,1); % 열벡터
    for i = N:-1:1
        X(i) = ( G(i,N+1) - G(i,i+1:N) * X(i+1:N) ) / G(i,i);
    end
        % 결과 출력
    disp('해 결과:');
    disp(X);

end

function matrix = pivot(matrix, row, N)
    % Scaling
    tColumn = zeros(1, N);
    for i = row:N
        M = max(abs(matrix(i,1:N)));
        tColumn(i) = abs(matrix(i,row) / M);
    end
    % Pivoting
    pivot_row = find(max(tColumn) == tColumn);
    if pivot_row ~= row
        pivot_vector = matrix(pivot_row,:);
        matrix(pivot_row,:) = matrix(row,:);
        matrix(row,:) = pivot_vector;
    end
end
```

```
NA13_2_GaussEliminationImproved();
```

```
해 결과:
    3.0000
```

```
   -2.5000
    7.0000
```

```
function NA13_2_GaussEliminationImproved2()
    G = [ 3, -0.1, -0.2, 7.85;
          0.3, -0.2, 10, 71.4;
          0.1, 7, -0.3, -19.3];
    N = 3;

    %% 전진 소거
    for i = 1:N-1
        G = pivot(G, i, N); % pivoting
        for j = i+1:N
            G(j,:) = G(j,:) - G(j,i) / G(i,i) * G(i,:);
        end
    end

    %% 후진 대입
    X = zeros(N,1); % 열벡터
    for i = N:-1:1
        X(i) = ( G(i,N+1) - G(i,i+1:N) * X(i+1:N) ) / G(i,i);
    end

    % 결과 출력
    disp('해 결과:');
    disp(X);
end

function matrix = pivot2(matrix, row, N)
    % Scaling
    tColumn = zeros(1, N);
    for i = row:N
        M = max(abs(matrix(i,1:N)));
        tColumn(i) = abs(matrix(i,row) / M);
    end

    % 가장 큰 값의 인덱스를 사용하여 피벗을 설정
    [~, pivot_row] = max(tColumn(row:N));
    pivot_row = pivot_row + row - 1; % 실제 행 인덱스를 맞추기 위해 조정

    if pivot_row ~= row
        pivot_vector = matrix(pivot_row,:);
        matrix(pivot_row,:) = matrix(row,:);
        matrix(row,:) = pivot_vector;
    end
end
```

```
NA13_2_GaussEliminationImproved2();
```

```
해 결과:
    3.0000
   -2.5000
    7.0000
```

개선된 코드(스케일링 코드부분)

```
function NA13_2_GaussEliminationImproved3()
    % 입력 행렬 (계수 행렬과 상수항 포함)
    G = [ 3, -0.1, -0.2, 7.85;
          0.3, -0.2, 10, 71.4;
          0.1, 7, -0.3, -19.3];
    N = size(G, 1);  % 행렬 크기 자동 결정

    %% 전진 소거
    for i = 1:N-1
        G = pivot(G, i, N); % pivoting
        for j = i+1:N
            G(j,:) = G(j,:) - G(j,i) / G(i,i) * G(i,:);  % 상부 삼각 행렬화
        end
    end

    %% 후진 대입
    X = zeros(N, 1);  % 해를 저장할 벡터
    for i = N:-1:1
        X(i) = (G(i, end) - G(i, i+1:N) * X(i+1:N)) / G(i, i);
    end

    % 결과 출력
    disp('해 결과:');
    disp(X);
end

function matrix = pivot3(matrix, row, N)
    % 개선된 스케일링 및 피벗 선택
    tColumn = abs(matrix(row:N, row)) ./ max(abs(matrix(row:N, 1:N)), [], 2);
    tColumn(isnan(tColumn)) = 0;  % NaN 값 발생 시 0으로 처리

    % 가장 큰 스케일링 값에 대한 피벗 행 찾기
    [~, pivot_row] = max(tColumn);
    pivot_row = pivot_row + row - 1;  % 실제 행 인덱스를 맞추기 위해 조정

    % 피벗 행이 현재 행과 다르면 행 교환
    if pivot_row ~= row
        pivot_vector = matrix(pivot_row, :);
        matrix(pivot_row, :) = matrix(row, :);
        matrix(row, :) = pivot_vector;
    end
```

```
end
```

```
NA13_2_GaussEliminationImproved3()
```

```
해 결과:
    3.0000
   -2.5000
    7.0000
```

```
G = [3, -0.1, -0.2, 7.85;
     0.1, 7, -0.3, -19.3;
     0.3, -0.2, 10, 71.4];

N = 3;
for i = 1:N
    if(G(i,i) <= 1e-10 ), fprintf("singular\n"), break, end
    G(i,:) = G(i,:) / G(i,i);  % 정규화
    for j = 1:N
        if (i ~= j)
            G(j,:) = G(j,:) - G(i,:) * G(j,i);
        end
    end
end
```

```
function GaussJordan()
    % 입력 행렬 (계수 행렬과 상수항 포함)
    G = [3, -0.1, -0.2, 7.85;
         0.1, 7, -0.3, -19.3;
         0.3, -0.2, 10, 71.4];

    N = 3;  % 행렬의 크기

    %% Gauss-Jordan 소거법 적용
    for i = 1:N
        % 특이 행렬 체크: 피벗 요소가 0 또는 0에 가까운지 확인
        if(G(i,i) <= 1e-10 )
            fprintf("행렬이 특이행렬입니다. 계산을 중지합니다.\n");
            return;
        end

        % 피벗 행 정규화
        G(i,:) = G(i,:) / G(i,i);  % 피벗 요소를 1로 만듦

        % 다른 행들의 소거
        for j = 1:N
            if (i ~= j)
                G(j, i:N+1) = G(j, i:N+1) - G(i, i:N+1) * G(j, i); %최적화코드
            end
```

```
        end
    end

    %% 결과 출력
    fprintf('최종 행렬 (단위 행렬화된 형태):\n');
    disp(G);  % 결과 행렬을 출력

    % 해 계산 (G의 마지막 열이 해를 나타냄)
    X = G(:, end);  % 마지막 열이 해를 의미
    fprintf('연립 방정식의 해 (x1, x2, x3):\n');
    disp(X);  % 해를 출력
end
```

```
GaussJordan()
```

```
최종 행렬 (단위 행렬화된 형태):
    1.0000         0         0    3.0000
         0    1.0000         0   -2.5000
         0         0    1.0000    7.0000

연립 방정식의 해 (x1, x2, x3):
    3.0000
   -2.5000
    7.0000
```