Project C : Solution of Tridiagonal Systems

2016060082 이지현
2017004002 박성진
2017004093 양준우

1. Gaussian elimination without pivoting

① Matlab code

```
% Gaussian elimination without pivoting

clear all;
clc;
format long;

n = input('input a degree of tridiagonal matrix: ');
diag = repelem(1,n); % diagonal elements vector
sub = repelem(-5,n-1); % subdiagonal elements vector
sup = repelem(2,n-1); % superdiagonal elements vector
b = repelem(-2,n);
b(1) = 3;
b(n) = -4; % b vector
rho(1) = 5; % Growth factor

for i = 2:n;
    diag(i) = diag(i) - (sub(i-1)/diag(i-1))*sup(i-1);
    b(i) = b(i) - (sub(i-1)/diag(i-1))*b(i-1);
    rho(i) = max([max(abs(sub)), max(abs(diag)), max(abs(sup))]);
end

x(n) = b(n)/diag(n);
for i = n-1:-1:1;
    x(i) = (b(i)-sup(i)*x(i+1))/diag(i);
end

A = gallery('tridiag',repelem(0,n-1),diag,sup);
full(A)
x'
error = x' - 1
growth_factor = max(rho)/rho(1)
norm = max(sum(abs(A')))*max(sum(abs(inv(A))'))
```

Pivoting을 사용하지 않는 가우스 소거법은 총 $3n-2$개의 저장공간이 필요하다. 이는 벡터 방식으로 계산을 할 때 0을 값으로 가지는 원소들을 애초에 저장하지 않기 때문이다. 우리가 사용하는 A라는 tridiagonal matrix는 대각성분 총 n개, subdiagonal n-1개, superdiagonal n-1 개만이 0이 아니기 때문에 총 $3n-2$개의 저장공간을 사용한다. 반면에 행렬을 사용하여 계산을 하는 경우에는 0을 값으로 가지는 원소들도 저장을 해야하기 때문에 총 n^2개만큼의 저장공간이 필요하다. 따라서 행렬로 계산하는 것은 보다 저장공간의 효율성 측면에서 비효율적임을 알 수 있다.

2. Gaussian elimination with pivoting

① Matlab code

```
% Gaussian elimination with partial pivoting

clear all;
clc;
format long;

n = input('input a degree of tridiagonal matrix: ');
diag = repelem(1,n); % diagonal elements vector
sub = repelem(-5,n-1); % subdiagonal elements vector
sup = repelem(2,n-1); % superdiagonal elements vector
supsup = repelem(0,n-2); % super superdiagonal elements vector
b = repelem(-2,n);
b(1) = 3;
b(n) = -4; % b vector
rho(1) = 5; % Growth factor
```

```
for i = 2:n;
    if abs(diag(i-1)) < abs(sub(i-1));
        sub(i-1) = diag(i-1); diag(i-1) = -5;
        diag(i) = sup(i-1); sup(i-1) = 1;
        K = b(i-1); b(i-1) = b(i); b(i) = K;
        if i ~= n;
            sup(i) = supsup(i-1); supsup(i-1) = 2;
        end
    end
    diag(i) = diag(i) - (sub(i-1)/diag(i-1))*sup(i-1);
    if i ~= n;
        sup(i) = sup(i) - (sub(i-1)/diag(i-1))*supsup(i-1);
    end
    b(i) = b(i) - (sub(i-1)/diag(i-1))*b(i-1);
    sub(i-1) = 0;
    rho(i)     =     max([max(abs(sub)),     max(abs(diag)),     max(abs(sup)),
max(abs(supsup))]);
end

x(n) = b(n)/diag(n);
x(n-1) = (b(n-1) - sup(n-1)*x(n))/diag(n-1);
for i = n-2:-1:1;
    x(i) = (b(i) - sup(i)*x(i+1) - supsup(i)*x(i+2))/diag(i);
end

r = repelem(0,n);
r(1) = -5;
c = repelem(0,n);
c(1) = -5; c(2) = 1; c(3) = 2;
A = toeplitz(r,c);
A(n,n) = diag(n)
A = sparse(A);
x'
error = x' - 1
growth_factor = max(rho)/rho(1)
norm = max(sum(abs(A')))*max(sum(abs(inv(A))'))
```

Pivoting을 사용하지 않은 가우스 소거법은 벡터 방식으로 계산했을 때 $3n-2$개 만큼의 저장공간을 사용하지만 partial pivoting 방법을 사용하면 기존의 $3n-2$개의 저장공간과 pivoting을 사용할 때 필요한 $n-2$개만큼의 0벡터들과 1개의 임시 공간(K)이 필요하므로 총 $4n-3$개 만큼의 저장공간이 필요하다. 이는 Matlab 코드를 보면 알 수 있다.

3. Result

① 저장공간의 효율성 비교

	G.E w/o pivoting	G.E w/ pivoting	using matrix
storage	$3n-2$	$4n-3$	n^2
$n=20$	58	77	400
$n=50$	148	197	2500
$n=100$	298	397	10000

표 1 : 방법에 따른 저장공간

가우스소거법을 row operation을 이용하여 각 행의 모든 원소에 대해 연산을 하게 되면 행렬의 크기인 n^2 만큼의 저장공간을 모두 사용해야 한다. 하지만 tridiagonal matrix는 계산이 간편하기 때문에 행 연산이 아닌, 벡터로 저장하여 각 벡터의 원소들의 연산으로 같은 결과를 얻을 수 있다. 한편 pivoting을 하는 경우에는 superdiagonal 바로 위의 대각성분을 새로 지정해야 하므로 n-2개의 저장공간이 추가로 요구된다.

	G.E w/o pivoting	G.E w/ pivoting
$n=20$	14.5%	19.25%
$n=50$	5.92%	7.88%
$n=100$	2.98%	3.97%

표 2 : 행렬로 계산했을 때와의 저장공간 크기 비교

위 표는 행렬과 비교했을 때의 저장공간의 크기를 퍼센트 비율로 나타낸 것이다. 이번 프로젝트의 경우에서는 행렬로 계산했을 때보다 벡터로 계산을 했을 때가 최소 2.98% ~ 최대 19.25%의 저장공간을 사용해 x를 구할 수 있다. 위 표에서의 수치를 보면 행렬의 크기 n이 커질수록 저장공간의 크기가 확연하게 차이가 커지는 것을 알 수 있다. 따라서 큰 행렬을 계산할 때에는 벡터를 이용하여 계산을 하는 것이 효율성이 좋다는 것을 알 수 있다.

② Approximate x

아래의 행렬은 각각 $n=20$에서의 가우스 소거법을 pivoting을 하지 않았을 때와 했을 때의 결과와 에러값이다.

● G.E W/O pivoting

$$A=\begin{pmatrix} 1 & 2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 11 & 2 & & 0 & 0 & 0 \\ 0 & 0 & 1.90909 & & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ \vdots & & & & \ddots & & \vdots \\ 0 & 0 & 0 & & 3.72375 & 2 & 0 \\ 0 & 0 & 0 & & 0 & 3.68547 & 2 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 3.71337 \end{pmatrix}$$

$$x-\begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \\ 1 \\ 1 \end{pmatrix}=\begin{pmatrix} \text{error}(\times 10^{-13}) \\ 0 \\ 0 \\ -0.002220446049250 \\ \vdots \\ 0.099920072216264 \\ 0.044408920985006 \\ 0.224265050974282 \end{pmatrix}$$

● G.E W/ pivoting

$$A=\begin{pmatrix} -5 & 1 & 2 & \cdots\cdots & 0 & 0 & 0 \\ 0 & -5 & 1 & & 0 & 0 & 0 \\ 0 & 0 & -5 & & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ \vdots & & & & \ddots & & \vdots \\ 0 & 0 & 0 & & -5 & 1 & 2 \\ 0 & 0 & 0 & & 0 & -5 & 1 \\ 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0.007077 \end{pmatrix}$$

$$x-\begin{pmatrix}1\\1\\1\\\vdots\\1\\1\\1\end{pmatrix}=\begin{pmatrix}0\\0\\-0.002220446049250\\\vdots\\-0.145439216225896\\-0.065503158452884\\-0.328626015289046\end{pmatrix}$$

$\mathrm{error}(1.0\times10^{-13})$

$n=20$일 때, pivoting을 한 경우와 하지 않은 경우 모두 에러가 거의 0에 가까운 것을 확인할 수 있었다.

	w/o pivoting				w/ pivoting			
n	20	50	100	200	20	50	100	200
x_1	1	1	1	1	1	1	1	1
x_2	1	1	1	1	1	1	1	1
⋮								
x_{n-4}	1.0	1.0	1.0001	1.5×10^{9}	0.9999	0.9999	0.9998	-0.2×10^{10}
x_{n-3}	1.0	1.0	1.0001	1.2×10^{9}	0.9999	0.9999	0.9998	-0.2×10^{10}
x_{n-2}	1.0	1.0	1.0002	3.1×10^{9}	0.9999	0.9999	0.9996	-0.5×10^{10}
x_{n-1}	1.0	1.0	1.0001	1.4×10^{9}	0.9999	0.9999	0.9998	-0.2×10^{10}
x_n	1.0	1.0	1.0006	7.1×10^{9}	0.9999	0.9999	0.9990	-1.0×10^{10}

표 3 : 피벗팅을 하지 않았을 때와 했을 때 추정된 해

표 3은 Matlab 코드를 통해 얻은 행렬의 크기(n)에 따른 해 x 벡터의 값이다. 표를 보면 알 수 있듯이 n이 작을 때는 해를 잘 추정한 모습이 보인다. 하지만 pivoting을 했을 때나 하지 않았을 때나 $n=100$일 때부터는 오차가 $n=20,50$에 비해 크게 나타남을 확인할 수 있다. 이를 통해 n이 커질수록 오차가 더 커질 수도 있다는 추론을 하게 되었고, $n=200$인 경우를 확인해본 결과, 오차가 10^9보다 크게 나타남을 확인할 수 있었다.

4. Analysis

일반적으로 피벗을 하지 않고 가우스 소거법을 사용하면 symmetric positive definite와 strictly diagonally dominant의 경우가 아닌 경우에는 해의 수치적 불안정성을 가져온다. 이번 프로젝트에서 우리에게 주어진 행렬의 경우는 위 두 가지 경우에 포함되지 않기 때문에 피벗을 해야지만 에러가 줄어든다고 생각을 할 수 있다. 하지만 위 결과에 보면 알 수 있듯이 피벗을 한 경우에도 n이 커질수록 에러가 커진다는 것을 알 수 있다.

우리는 이런 현상을 행렬의 norm을 사용해 설명하고자 한다. $\varkappa(A)$는 행렬의 norm을 사용해 다음과 같이 정의할 수 있다.

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

$\varkappa(A)$가 의미하는 것은 결과적으로 $Ax = b$ 라는 선형방정식에서 A와 b에서의 에러가 얼마나 민감하게 해 x에 영향을 끼치는 정도이다.

b에서 δb만큼의 에러가 발생하여 x에서 δx만큼의 에러가 발생했다고 생각하면, 다음과 같은 식을 얻을 수 있다.

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

또한 $Ax = b$, $A\delta x = \delta b$ 이기 때문에, $\delta x = A^{-1}\delta b$가 유도된다. 따라서

$\| b \| = \| Ax \| \leq \| A \| \| x \| \cdots$ ①

$\| \delta x \| \leq \| A^{-1} \| \| \delta b \| \cdots$ ②

를 도출할 수 있고, ① 과 ②에 의해

$$\frac{\| \delta x \|}{\| x \|} \leq \| A \| \| A^{-1} \| \frac{\| \delta b \|}{\| b \|} = \varkappa(A) \frac{\| \delta b \|}{\| b \|}.$$

라는 식을 유도할 수 있다.

$\frac{\| \delta x \|}{\| x \|}$는 x의 relative error를 의미하고, $\frac{\| \delta b \|}{\| b \|}$는 상수 벡터인 b의 relative error를 의미한다. 따라서 위 식은 우리가 구하는 해의 에러가 $\varkappa(A)$와 $\frac{\| \delta b \|}{\| b \|}$의 곱보다 커질 수 없다는 것을 의미한다. 즉 $\varkappa(A)$가 커질수록 에러가 커진다는 것을 알 수 있다.

우리가 프로젝트에서 사용한 행렬의 n에 따른 $\varkappa(A)$의 값은 다음과 같다.

n	20	50	100	200
$\varkappa(A)$ w/o pivoting	19.83460	19.83466	19.83466	19.83466
$\varkappa(A)$ w/ pivoting	1.1304×10^{3}	9.3621×10^{6}	3.1668×10^{13}	3.6234×10^{26}

표 4 : 행렬의 크기(n)에 따른 $\varkappa(A)$

위 표에서의 pivoting을 했을 때의 $\varkappa(A)$ 값을 보면 n이 커질수록 매우 커지는 것을 확인할 수 있다. 이를 통해 pivoting을 해도 에러가 커짐을 설명할 수 있다.

5. Conclusion

이번 프로젝트를 통해 $Ax = b$라는 연립선형방정식의 해를 가우스소거법을 통해 해를 추정했고, 저장공간의 효율성과 해의 정확성에 대한 분석을 진행했다. 저장공간의 효율성 측면에서는 행렬을 이용할 때보다 벡터로 하는 것이 훨씬 효율적이며, pivoting을 할 때가 하지 않을 때보다 조금 더 많은 저장공간을 사용해야 함을 보였다. 또한 해를 추정한 결과, n의 크기가 작을 때는 비교적 해를 잘 추정했지만, n의 크기가 커질수록 $\varkappa(A)$가 커지기 때문에 오차가 커지는 것을 확인할 수 있었다.