

Project C – Solution of Tridiagonal Systems

Junwoo Yang

June 3, 2019

1 Gaussian elimination without pivoting

1.1 MATLAB code

```
1 clear all; clc; format long;
2
3 n = input('input a degree of tridiagonal matrix: ');
4 diag = repelem(1,n); % diagonal elements vector
5 sub = repelem(-5,n-1); % subdiagonal elements vector
6 sup = repelem(2,n-1); % superdiagonal elements vector
7 b = repelem(-2,n);
8 b(1) = 3;
9 b(n) = -4; % b vector
10 rho(1) = 5; % Growth factor
11
12 for i = 2:n;
13     diag(i) = diag(i) - (sub(i-1)/diag(i-1))*sup(i-1);
14     b(i) = b(i) - (sub(i-1)/diag(i-1))*b(i-1);
15     rho(i) = max([max(abs(sub)), max(abs(diag)), max(abs(sup))]);
16 end
17
18 x(n) = b(n)/diag(n);
19 for i = n-1:-1:1;
20     x(i) = (b(i)-sup(i)*x(i+1))/diag(i);
21 end
22
23 A = gallery('tridiag',repelem(0,n-1),diag,sup);
24 full(A)
25 x'
26 error=x'-1
27 growth_factor = max(rho)/rho(1)
28 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 A 라는 tridiagonal matrix는 대각성분 총 n 개, subdiagonal $n - 1$ 개, 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^2 개만큼의 저장공간이 필요하다. 따라서 행렬로 계산하는 것은 보다 저장공간의 효율성 측면에서 비효율적임을 알 수 있다.

2 Gaussian elimination with pivoting

2.1 MATLAB code

```

1 clear all; clc; format long;
2
3 n = input('input a degree of tridiagonal matrix: ');
4 diag = repelem(1,n); % diagonal elements vector
5 sub = repelem(-5,n-1); % subdiagonal elements vector
6 sup = repelem(2,n-1); % superdiagonal elements vector
7 supsup = repelem(0,n-2); % super superdiagonal elements vector
8 b = repelem(-2,n);
9 b(1) = 3;
10 b(n) = -4; % b vector
11 rho(1) = 5; % Growth factor
12
13 for i = 2:n;
14     if abs(diag(i-1)) < abs(sub(i-1));
15         sub(i-1) = diag(i-1); diag(i-1) = -5;
16         diag(i) = sup(i-1); sup(i-1) = 1;
17         K = b(i-1); b(i-1) = b(i); b(i) = K;
18         if i ~= n;
19             sup(i) = supsup(i-1); supsup(i-1) = 2;
20         end
21     end
22     diag(i) = diag(i) - (sub(i-1)/diag(i-1))*sup(i-1);
23     if i ~= n;
24         sup(i) = sup(i) - (sub(i-1)/diag(i-1))*supsup(i-1);
25     end
26     b(i) = b(i) - (sub(i-1)/diag(i-1))*b(i-1);
27     sub(i-1) = 0;
28     rho(i) = max([max(abs(sub)), max(abs(diag)), max(abs(supsup))]);
29 end
30

```

```

31 x(n) = b(n)/diag(n);
32 x(n-1) = (b(n-1) - sup(n-1)*x(n))/diag(n-1);
33 for i = n-2:-1:1;
34     x(i) = (b(i) - sup(i)*x(i+1) - supsup(i)*x(i+2))/diag(i);
35 end
36
37 r = repelem(0,n);
38 r(1) = -5;
39 c = repelem(0,n);
40 c(1) = -5; c(2) = 1; c(3) = 2;
41 A = toeplitz(r,c);
42 A(n,n) = diag(n)
43 A = sparse(A);
44 x'
45 error = x' - 1
46 growth_factor = max(rho)/rho(1)
47 norm = max(sum(abs(A')))*max(sum(abs(inv(A))))

```

Pivoting을 사용하지 않은 가우스 소거법은 벡터 방식으로 계산했을 때 $3n - 2$ 개 만큼의 저장 공간을 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

3.1 저장공간의 효율성 비교

	G. E. without pivoting	G.E. with 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

Table 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 $n - 2$ 개의 저장공간이 추가로 요구된다.

Table 2는 행렬과 비교했을 때의 저장공간의 크기를 퍼센트 비율로 나타낸 것이다. 이번 프로젝트의 경우에는 행렬로 계산했을 때보다 벡터로 계산을 했을 때가 최소 2.98% ~ 최대 19.25%의 저장공간을

	G. E. without pivoting	G.E. with pivoting
$n = 20$	14.5%	19.25%
$n = 50$	5.92%	7.88%
$n = 100$	2.98%	3.97%

Table 2: 행렬로 계산했을 때와의 저장공간 크기 비교

사용해 x 를 구할 수 있다. 위 표에서의 수치를 보면 행렬의 크기 n 이 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.

3.2 Approximate x

아래는 $n = 20$ 일 때, pivoting을 하지 않은 가우스 소거법의 결과와 error term이다.

$$A = \begin{pmatrix} 1 & 2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 11 & 2 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1.91 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 3.72 & 2 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 3.69 & 2 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 3.71 \end{pmatrix}, \quad x - \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -0.002 \\ \vdots \\ 0.099 \\ 0.044 \\ 0.224 \end{pmatrix}.$$

한편, pivoting을 해주었을 경우, 결과는 다음과 같다.

$$A = \begin{pmatrix} -5 & 1 & 2 & \cdots & 0 & 0 & 0 \\ 0 & -5 & 1 & \cdots & 0 & 0 & 0 \\ 0 & 0 & -5 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -5 & 1 & 2 \\ 0 & 0 & 0 & \cdots & 0 & -5 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0.007 \end{pmatrix}, \quad x - \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -0.002 \\ \vdots \\ -0.145 \\ -0.065 \\ -0.328 \end{pmatrix}.$$

$n = 20$ 일 때, pivoting을 한 경우와 하지 않은 경우 모두 에러가 거의 0에 가까운 것을 확인할 수 있다. Table 3은 MATLAB code를 통해 얻은 행렬의 크기 n 에 따른 해 x 벡터의 값이다. 표를 보면 알 수 있듯이 n 이 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 n 이 커질수록 오차가 더 커질 수도 있다는 추론을 하게 되었고, $n = 200$ 인 경우를 확인해본 결과, 오차가 10^9 보다 크게 나타남을 확인할 수 있었다.

n	G. E. without pivoting				G.E. with pivoting			
	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
\vdots								
x_{n-4}	1	1	1.0001	1.5×10^9	0.9999	0.9999	0.9998	-0.2×10^{10}
x_{n-3}	1	1	1.0001	1.2×10^9	0.9999	0.9999	0.9998	-0.2×10^{10}
x_{n-2}	1	1	1.0002	3.1×10^9	0.9999	0.9999	0.9996	-0.5×10^{10}
x_{n-1}	1	1	1.0001	1.4×10^9	0.9999	0.9999	0.9998	-0.2×10^{10}
x_n	1	1	1.0006	7.1×10^9	0.9999	0.9999	0.9990	-1.0×10^{10}

Table 3: pivoting을 하지 않았을 때와 했을 때 추정된 해

4 Analysis

일반적으로 피벗을 하지 않고 가우스 소거법을 사용하면 symmetric positive definite와 strictly diagonally dominant의 경우가 아닌 경우에는 해의 수치적 불안정성을 가져온다. 이 번 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 n 이 커질수록 에러가 커진다는 것을 알 수 있다. 우리는 이런 현상을 행렬의 norm을 사용해 설명하고자 한다. $\chi(A)$ 는 행렬의 norm을 사용해 다음과 같이 정의할 수 있다.

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

$\chi(A)$ 가 의미하는 것은 결과적으로 선형방정식 $Ax = b$ 에서 해 x 가 A 와 b 의 에러에 민감하게 반응하는 정도를 나타낸다. b 에서 δb 만큼의 에러가 발생하여 x 에서 δx 만큼의 에러가 발생했다고 생각하면, 다음과 같은 식을 얻을 수 있다.

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

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

$$\|b\| = \|Ax\| \leq \|A\| \|x\| \tag{1}$$

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

을 도출할 수 있고, (1)과 (2)에 의해 다음을 얻을 수 있다.

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

$\frac{\|\delta x\|}{\|x\|}$ 는 x 의 relative error, $\frac{\|\delta b\|}{\|b\|}$ 는 상수 벡터 b 의 relative error를 의미한다. 따라서 위 식은 우리가 구하는 해의 에러가 $\chi(A)$ 와 $\frac{\|\delta b\|}{\|b\|}$ 의 곱보다 커질 수 없다는 것을 의미한다. 즉 $\chi(A)$ 가 커질수록 에러가 커진다는 것을 알 수 있다. 우리가 프로젝트에서 사용한 행렬의 n 에 따른 $\chi(A)$ 의 값은 Table 4와 같다. pivoting을 했을 때의 $\chi(A)$ 값을 보면 n 이 커질수록 매우 커지는 것을 확인할 수 있다. 이를 통해 pivoting을 해도 에러가 커짐을 설명할 수 있다.

n	20	50	100	200
$\chi(A)$ without pivoting	19.83460	19.83466	19.83466	19.83466
$\chi(A)$ with pivoting	1.1304×10^3	9.3621×10^6	3.1668×10^{13}	3.6234×10^{26}

Table 4: 행렬의 크기 n 에 따른 $\chi(A)$

5 Conclusion

가우스소거법을 통해 연립선형방정식 $Ax = b$ 의 해를 추정했고, 저장공간의 효율성과 해의 정확성에 대한 분석을 진행했다. 저장공간의 효율성 측면에서 는 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, n 의 크기가 작을 때는 비교적 해를 잘 추정했지만, n 의 크기가 커질수록 $\chi(A)$ 가 커지기 때문에 오차가 커지는 것을 확인할 수 있었다.