Matthew LeDuc
APPM 5600
Fall 2023
HW 6

1 Question 1

The condition number k of solving the problem $Ax = b$ in the presence of perturbations δb in b is given by

$$k = \frac{||\delta x||}{||x||} \frac{||b||}{||\delta b||}$$

Since $x + \delta x = A^{-1}(b + \delta b)$, we cna write that

$$k = \frac{||\delta x||}{||x||} \frac{||Ax||}{||\delta b||} \leq ||A|| \frac{||x||}{||x||} \frac{||\delta x||}{||\delta b||} = ||A|| \frac{||\delta x||}{||\delta b||}$$

Likewise $\delta b = A\delta x$, so we can say that

$$k \leq ||A|| \left|\left|A^{-1}\right|\right| = \kappa(A)$$

2 Question 2

(a) We are asked to find an expression for δx given a perturbation δb. In this case, we have that

$$x + \delta x = A^{-1}(b + \delta b)x + A^{-1}\delta b$$

so $\delta x = A^{-1}\delta b$. Using this, we can write that

$$\delta x = \begin{pmatrix} 1 - 10^{10} & 10^{10} \\ 1 + 10^{10} & -10^{10} \end{pmatrix} \begin{pmatrix} \Delta b_1 \\ \Delta b_2 \end{pmatrix}$$

So, doing the multiplication,

$$\delta x = \begin{pmatrix} \Delta b_1 + 10^{10}(\Delta b_1 - \Delta b_2) \\ \Delta b_1 + 10^{10}(\Delta b_2 - \Delta b_1) \end{pmatrix}$$

(b) Using MATLAB, I found that $\kappa(A) = 2 \times 10^{10}$

(c) The relative error is given by $\frac{||\delta x||}{||x||}$. If $\Delta b_i = O(10^{-5})$ then the difference is approximately 10^{-5}, and so

$$||\delta x||^2 \approx 2(10^{-5} + 10^5)^2$$

, and so the relative error is given by

$$\frac{||\delta x||}{||x||} \approx 10^5$$

since $10^{-5} << 10^5$. This means that in this case our solution is not accurate to any number of decimal places, and in fact we lose close to the maximum amount of accuracy in the solution given the size of the perturbation. However, if $\Delta b_1 - \Delta b_2$ is very small, we can find that

$$\frac{||\delta x||}{||x||} \approx 10^{-5}$$

so we have 5 digits of accuracy.

3 Question 3

(a) We can begin by writing the matrix equations as

$$\begin{pmatrix} a_1 & c_2 & 0 \\ b_2 & a_2 & c_2 \\ 0 & b_3 & a_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \tag{3.1}$$

We then proceed with the matrix multiplication: this implies that $u_{11} = a_1$ and that $u_{11}l_{21} = b_2 \implies l_{21} = \frac{b_2}{a_1}$. From this we also get that $l_{31} = 0$. Moving on to the next column, we follow the same process to see that

$$l_{32} = \frac{b_3}{a_2 - c_1 l_{21}},\ u_{22} = a_2 - c_1 l_{21},\ u_{12} = c_1$$
$$u_{13} = 0,\ u_{23} = c_2,\ u_{33} = a_3 - l_{32}u_{23},$$

(b) We can see a pattern emerging in the answer to part A, where the entries of L are given by

$$\begin{aligned} l_{n,n-1} &= \frac{b_n}{u_{n-1,n-1}} \\ l_{n,n-2} &= l_{n,n-3} = ... = 0 \end{aligned} \tag{3.2}$$

and the entries of U are given by

$$\begin{aligned} u_{1,1} &= a_1 \\ u_{n,n} &= a_n - l_{n,n-1}u_{n-1,n} \\ u_{n,n+1} &= c_n \\ u_{n,n-2} &= u_{n,n-3} = ... = 0 \end{aligned} \tag{3.3}$$

This can be verified by induction: We already showed that the pattern holds for $n = 3$ so suppose that it holds for $n = k$. Then the LU-decomposition of the first $k \times k$ minor of A follows the pattern. Carrying through the matrix multiplication, since we have that L and U must be lower and upper triangular and A is tridiagonal, so the pattern continues to hold.

(c) For the first row, we do not do anything. For the k^{th} row, we can hard-code the element below the diagonal to 0, then there is one multiplication and one subtraction to perform to alter the on-diagonal element. This part of the process requires $2(n-1)$ flops. The same thing must be done to the right-hand side of the equation, requiring $2(n-1)$ flops again, so we end with $4n - 4$ flops.

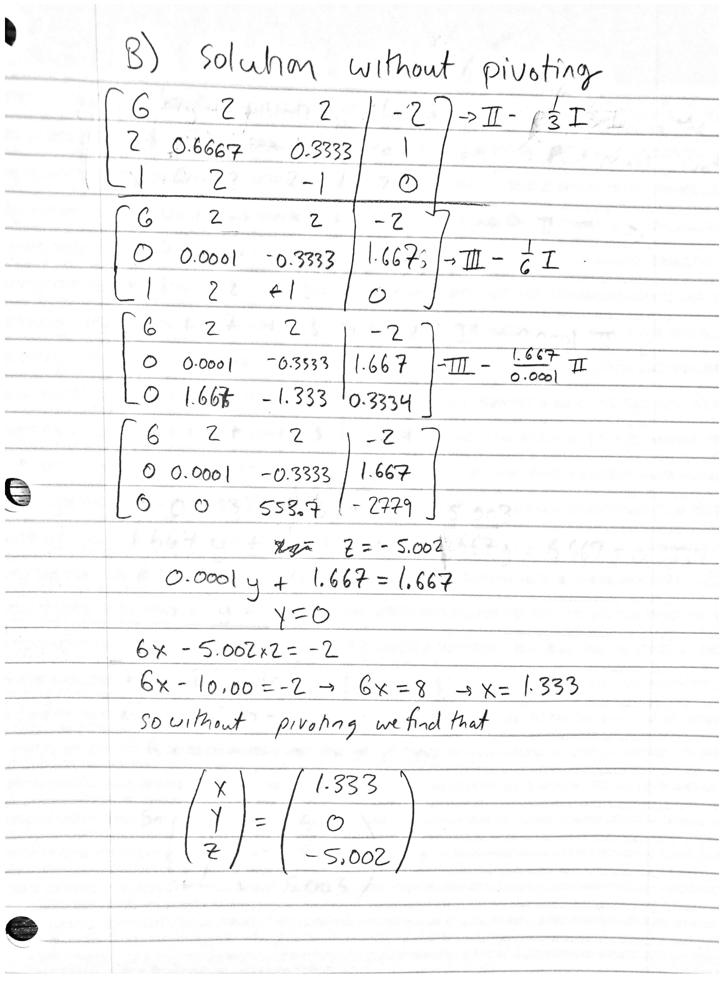
4 Question 4

(a) Given the solution $\vec{x_*} = (2.6, -3.8, 5)^T$, in the first row we get that

$$\begin{aligned} 6(2.6) + 2(-3.8) + 2(-5) &= 15.6 - 7.6 - 10 = -2 \\ 2(2.6) + \frac{2}{3}(-3.8) + \frac{1}{3}(-5) &= 5.2 - \frac{7.6}{3} - \frac{5}{3} = 1 \\ 2.6 - 7.6 + 5 &= 0 \end{aligned} \tag{4.1}$$

so the given $\vec{x_*}$ is the solution to the system of equations.

(b) Below is the solution for part B.



B) Solution without pivoting

$$\left[\begin{array}{ccc|c} 6 & 2 & 2 & -2 \\ 2 & 0.6667 & 0.3333 & 1 \\ 1 & 2 & -1 & 0 \end{array}\right] \rightarrow II - \frac{1}{3} I$$

$$\left[\begin{array}{ccc|c} 6 & 2 & 2 & -2 \\ 0 & 0.0001 & -0.3333 & 1.667 \\ 1 & 2 & +1 & 0 \end{array}\right] \rightarrow III - \frac{1}{6} I$$

$$\left[\begin{array}{ccc|c} 6 & 2 & 2 & -2 \\ 0 & 0.0001 & -0.3333 & 1.667 \\ 0 & 1.667 & -1.333 & 0.3334 \end{array}\right] \rightarrow III - \frac{1.667}{0.0001} II$$

$$\left[\begin{array}{ccc|c} 6 & 2 & 2 & -2 \\ 0 & 0.0001 & -0.3333 & 1.667 \\ 0 & 0 & 553.7 & -2779 \end{array}\right]$$

$z = -5.002$

$0.0001y + 1.667 = 1.667$

$y = 0$

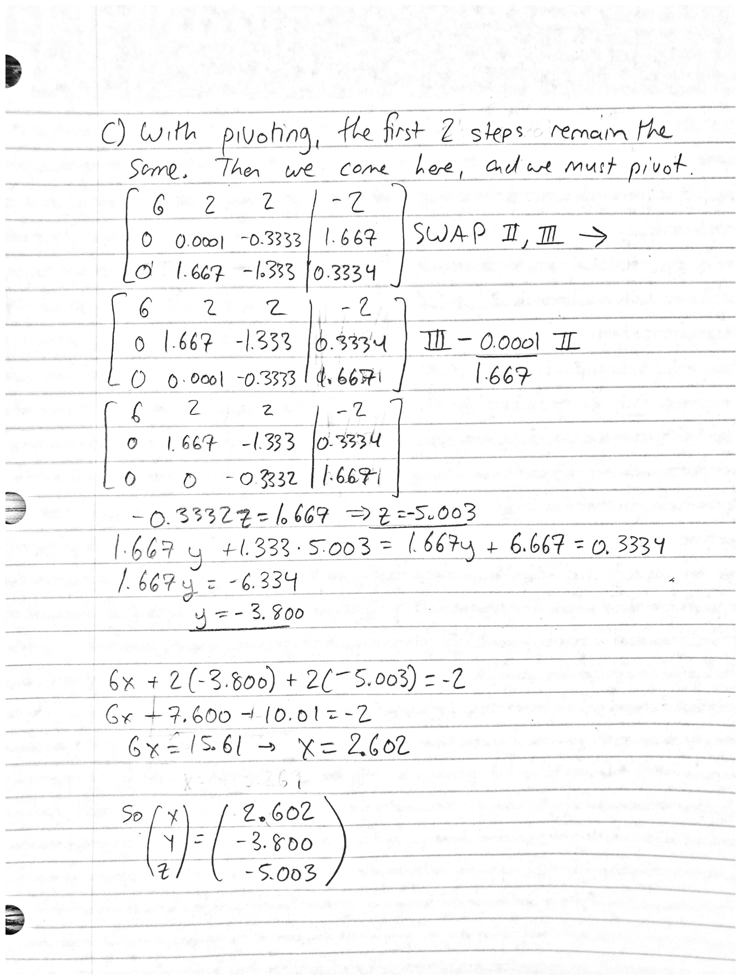
$6x - 5.002 \times 2 = -2$

$6x - 10.00 = -2 \rightarrow 6x = 8 \rightarrow x = 1.333$

so without pivoting we find that

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1.333 \\ 0 \\ -5.002 \end{pmatrix}$$

(c) Below is the solution for part C.



C) With pivoting, the first 2 steps remain the same. Then we come here, and we must pivot.

$$\left[\begin{array}{ccc|c} 6 & 2 & 2 & -2 \\ 0 & 0.0001 & -0.3333 & 1.667 \\ 0 & 1.667 & -1.333 & 0.3334 \end{array}\right] \text{SWAP II, III} \rightarrow$$

$$\left[\begin{array}{ccc|c} 6 & 2 & 2 & -2 \\ 0 & 1.667 & -1.333 & 0.3334 \\ 0 & 0.0001 & -0.3333 & 1.6671 \end{array}\right] \text{III} - \frac{0.0001}{1.667}\,\text{II}$$

$$\left[\begin{array}{ccc|c} 6 & 2 & 2 & -2 \\ 0 & 1.667 & -1.333 & 0.3334 \\ 0 & 0 & -0.3332 & 1.6671 \end{array}\right]$$

$-0.3332z = 1.669 \Rightarrow \underline{z = -5.003}$

$1.667y + 1.333 \cdot 5.003 = 1.667y + 6.667 = 0.3334$

$1.667y = -6.334$

$\underline{y = -3.800}$

$6x + 2(-3.800) + 2(-5.003) = -2$

$6x + 7.600 + 10.01 = -2$

$6x = 15.61 \rightarrow x = 2.602$

So $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2.602 \\ -3.800 \\ -5.003 \end{pmatrix}$

(d) The pivoting method is much more accurate. The relative error for the pivoting calculation is approximately 5×10^{-4} which when keeping 4 digits is probably about as good as it gets practically. However, without pivoting the relative error is about 0.58, which is much much worse.

5 Question 5

(a) The Gauss-Jacobi method converged in 37 iterations to

$$x^* \approx \begin{pmatrix} 1.16666649 \\ 1.20833310 \\ 1.45833310 \\ 1.45833310 \\ 1.20833310 \\ 1.16666649 \end{pmatrix}$$

(b) The Gauss-Seidel method converged in 21 iterations to

$$x^* \approx \begin{pmatrix} 1.16666658 \\ 1.20833324 \\ 1.45833326 \\ 1.45833326 \\ 1.20833328 \\ 1.16666663 \end{pmatrix}$$

(c) The SOR method with $\omega = 1.6735$ converged in 45 iterations to

$$x^* \approx \begin{pmatrix} 1.16666659 \\ 1.20833337 \\ 1.45833331 \\ 1.45833325 \\ 1.20833337 \\ 1.16666663 \end{pmatrix}$$

(d) The Gauss-Seidel method converges the fastest. According to MATLAB, $||M^{-1}N|| \approx$ 0.683 for Gauss-Jacobi, 0.6123 for Gauss-Seidel, and 1.0724 for SOR. From this, I would expect that Gauss-Seidel would usually converge the fastest, however there may be certain b for which another method may converge faster.

(e) Using the error bound

$$||x_{k+1} - x|| \leq \frac{c}{1-c} ||x_{x+1} - x_k||$$

the approximate errors are given by the following table: ‘

Method	Error
Gauss-Jacobi	$10^{-6.28}$
Gauss-Seidel	$10^{-6.75}$
SOR	$10^{-6.11}$

This implies that the Gauss-Seidel method is the most accurate, followed by Gauss-Jacobi and SOR.

(f) I ran the code for 51 values of ω between 0.6735 and 2.6735 for a maximum of 10^4 iterations. The number of iterations to convergence is plotted against the value of ω in Figure 5.1. It seems as though the algorithm converges fastest for $\omega \approx 1.21$, but as $\omega \to 2$ it takes longer and longer to converge, and fails to do so once $\omega \geq 2$.

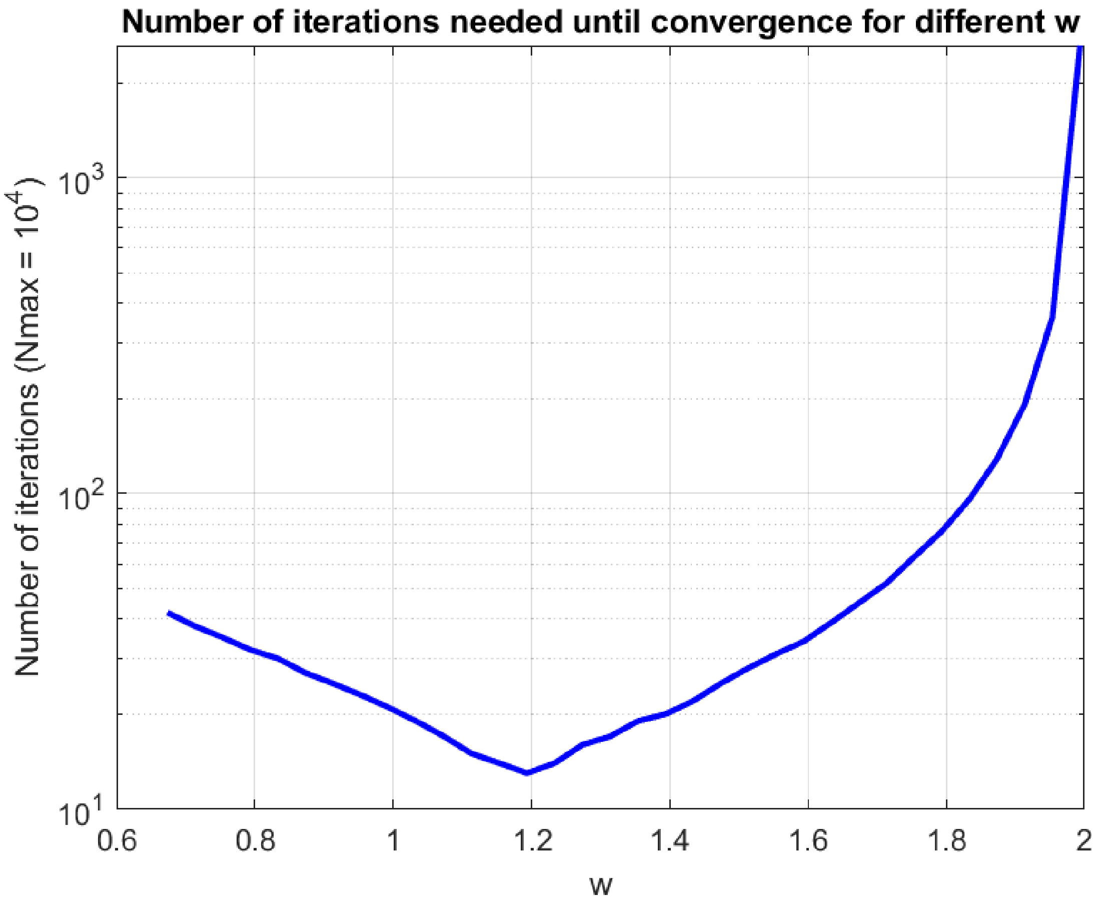


Figure 5.1: Number of iterations for convergence vs ω

5.1 Code

5.1.1 Run script

```
%%
clear variables; close all;

A = diag(2*ones([1,6]));%Building A
%Start with a 2 on the diagonal so we can add L+L^T
for ii = 1:5
    A(ii,ii+1)=-1;
    if ii<=3
```

```
        A(ii,ii+3)=-1;
    end
end
A = A+A.';
b = [2,1,2,2,1,2].';
x = ones(6,1);

tol = 1e-7;
maxIter = 10000;
%% Gauss-Jacobi method
[xJ,errJ,cJ,normJ] = stationaryIters(A,b,x,tol, maxIter, 'jacobi');
fprintf('Jacobi finished in %d iterations, operator norm is %.4f\n',cJ,normJ);

%% Gauss-Seidel
[xG,errG,cG,normG] = stationaryIters(A,b,x,tol, maxIter, 'gauss-seidel');
fprintf('Gauss-Seidel finished in %d iterations, operator norm is %.4f\n',cG,normG);

%% SOR
w = 1.6735;
[xS,errS,cS,normS] = stationaryIters(A,b,x,tol, maxIter, 'SOR', w);
fprintf('SOR finished in %d iterations, operator norm is %.4f\n\n',cS,normS);
fprintf('log10(Error approximations):\n');
fprintf('Jacobi: %.6f\n', log10(errJ));
fprintf('Gauss-Seidel: %.6f\n', log10(errG));
fprintf('SOR: %.6f\n', log10(errS));

nws = 51;%Number of w values to test
ws = linspace(w-1, w+1, nws);
solns = zeros( 6, nws );
errs = zeros(1,nws);
iters = zeros(1,nws);
for ii = 1:nws
    [solns(:,ii),tmp1,iters(ii),tmp3,errs(ii)] = stationaryIters(A,b,x,tol, maxIter, 'SOR',
end

figure;semilogy(ws(errs==0), iters(errs==0), 'b', 'linewidth',2)
xlabel('w')
ylabel('Number of iterations (Nmax = 10^4)')
title('Number of iterations needed until convergence for different w')
grid on
saveas( gcf, 'sor_wvsiters.png' );
savefig( gcf, 'sor_wvsiters.fig'  )
```

5.1.2 Stationary Iterations Code

```
function [x,errEst, cnt, MiNnorm, errMsg] = stationaryIters( A,b, x0, tol, maxIters, metho
%Stationary iterations using Jacobi, Gauss-Seidel, or successive
%over-relaxation methods to solve Ax=b for x
%Inputs:
%   A: The matrix, Ax=b
%   b: The y-value. Ax=b
%   x0: Initial guess.
%   tol: Relative error tolerance
%   maxIters: Max number of iterations allowed
%   method: Which algorithm to use. One of 'jacobi', 'gauss-seidel', or
%       'sor'
%   varargin: If using SOR, the value of \omega
%Outputs:
%   x: Estimate of the solution
%   errEst: Error estimate, c/(1-c)*norm(dx)
%   cnt: Number of iterations
%   MiNnorm: operator norm of M^{-1}N
%   errMsg: 0 if successful, 1 if ran out of iterations, 2 if returned NaN
[n,m] = size(A,[1,2]);
if n~=m
   error( 'A is not square' )
end
if strcmpi(method, 'jacobi')%% Pick an algorithm
   M = diag(A).*eye(n);
   L = tril(A)-M;
   U = triu(A)-M;
   N = -(L+U);
elseif strcmpi(method, 'gauss-seidel')
   M = tril(A);
   N = M-A;
elseif strcmpi(method, 'SOR')
    if ~isempty(varargin{1})
        w = varargin{1};
    else
        w = 1.6735;
    end
    D = diag(A).*eye(n);
    L = tril(A)-D;
    U = triu(A)-D;
    M = D+w*L;
    N = (1-w)*D-w*U;
    b = w*b;
end
Mi = inv(M);
MiNnorm = norm(Mi*N, 2);
del = tol+1;
cnt = 0;
while cnt<maxIters && del>tol %Implementing the iteration
    x1 = Mi*(N*x0+b);
    del = norm(x1-x0)/norm(x0);
    cnt=cnt+1;
    xn1 = x0;
    x0 = x1;
end
errMsg = 0;
x=x0;
eigVals = eig( Mi*N );%% Getting teh error estimate
spectralRad = max(abs(eigVals));
errEst = abs(spectralRad/(1-spectralRad))*norm( x0-xn1 );
if cnt<maxIters && all(isfinite(x))
    errMsg = 0;
elseif all(isfinite(x))
```

```
    errMsg = 1;
else
    errMsg = 2;
end
end
```