Assignment 3

COSC2500
Ryan White
44990392

21st of September 2020

R3.1

a. Using the Matlab Code shown in Appendix 1 (specifically under the section `PART A`), the solutions to the systems of linear equations were as follows:

i.

$$\begin{aligned} 2x - 2y - z &= -2 \\ 4x + y - 2z &= 1 \\ -2x + y - z &= -3 \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

ii.

$$\begin{aligned} x + 2y - z &= 2 \\ 3y + z &= 4 \\ 2x - y + z &= 2 \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

iii.

$$\begin{aligned} 2x + y - 4z &= -7 \\ x - y + z &= -2 \\ -x + 3y - 2z &= -6 \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \\ 2 \end{bmatrix}$$

b. Once again, the following answers were calculated using the code given in Appendix 1, under the section labeled `PART B`. For the various cases of matrix sizes, the solutions are

$$n = 2 \qquad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 6 \end{bmatrix}$$

$$n = 5 \qquad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 5 \\ -112 \\ 630 \\ -1120 \\ 630 \end{bmatrix}$$

$$n = 10 \qquad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix} = \begin{bmatrix} -9.997365 \\ 989.771861 \\ -23755.13378 \\ 240195.7143 \\ -1261048.597 \\ 3783198.501 \\ -6725765.490 \\ 7000357.238 \\ -3937735.418 \\ 923673.4085 \end{bmatrix}$$

R3.2

a. For $n = 6$, the condition number of matrix A was found to be approx 3.9142×10^7, meaning that, in the worst case scenario, the resultant answer would be 7 digits less accurate. Using the code shown in Appendix 2.1, the forward error was calculated as 5.4789×10^{-10}, and the final error magnification factor being 2.467488×10^6. This means that the actual result for x_c had a 6 digit accuracy loss, one digit more favourable than the calculated. condition number.
For $n = 10$, the condition number was calculated to be 6.6876×10^{13}. Using the same code as for $n = 6$, with the value for n changed, the forward error was then found to be 1.1×10^{-3}, and the error magnification factor calculated as 5.0095×10^{12}. Again, this is one digit more favourable than the condition number, although it is significantly less accurate than the case for $n = 6$ (12 vs 6 lost digits of accuracy).

b. The comparison of condition number, forward error, and magnification error factors, as well as resultant digits of accuracy, for the matrix elements having the value $A_{ij} = |i - j| + 1$, is shown in Table 1

Matrix Dimensions ($n \times n$)	Condition Number	Forward Error	Error Magnification Factor	Correct Digits
100	7.1389×10^3	1.3785×10^{-12}	6.208×10^3	13
200	2.8177×10^4	1.4360×10^{-11}	6.4672×10^4	12
300	6.3110×10^4	2.6077×10^{-11}	1.17440×10^5	11
400	1.1194×10^5	9.6179×10^{-11}	4.33152×10^5	11
500	1.7467×10^5	1.2001×10^{-10}	5.40480×10^5	11

Table 1: Comparison of Errors for Varying Matrix Dimensions

c. While all of the above answers have at least *some* correct digits, the $n = 10$ case for part a. had the least number of correct digits (4 correct digits). The next lowest number of correct digits corresponded to the $n = 6$ case of the same part, with only 10 correct digits. This is likely due to the matrix values in part a. having more computation involved than in part b.

R3.3

Table 2 compares some of the properties of Iterative methods of solving systems of linear equations:

Iterative Method	Coefficient Matrix Restrictions	Time	Convergence
Jacobi Method	Square	$O(n^2)$	Only if Strictly Diagonally Dominant (SDD)
Gauss-Seidel	Square	$O(n^2)$	Only if SDD
Successive Over-Relaxation	Square	$O(n^2)$	Only if SDD
Generalised Minimum Residual	Square	$O(n^2)$	-
BiConjugate Gradients	Square	$O(n^2)$	-
Conjugate Gradients Squared	Square	$O(n^2)$	-
Symmetric LQ	Square, Symmetric	$O(n^2)$	-
Minimum Residual	Square, Symmetric	$O(n^2)$	-

Table 2: Comparison of Iterative Methods

R3.4

For the two matrices (shown in Sauer Computer Problems 2.5 1. & 2.),

$$A_1 = \begin{bmatrix} 3 & -1 & & & \\ -1 & 3 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 3 & -1 \\ & & & -1 & 3 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 2 & 1 & & & \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ & & & 1 & 2 \end{bmatrix}$$

with RHS vectors

$$b_1 = \begin{bmatrix} 2 \\ 1 \\ \vdots \\ 1 \\ 2 \end{bmatrix} \qquad b_2 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ -1 \end{bmatrix}$$

the computation time for solving with matrix dimensions $n \times n$ were calculated with the code shown in Appendix 3, and are seen in Figure 1.

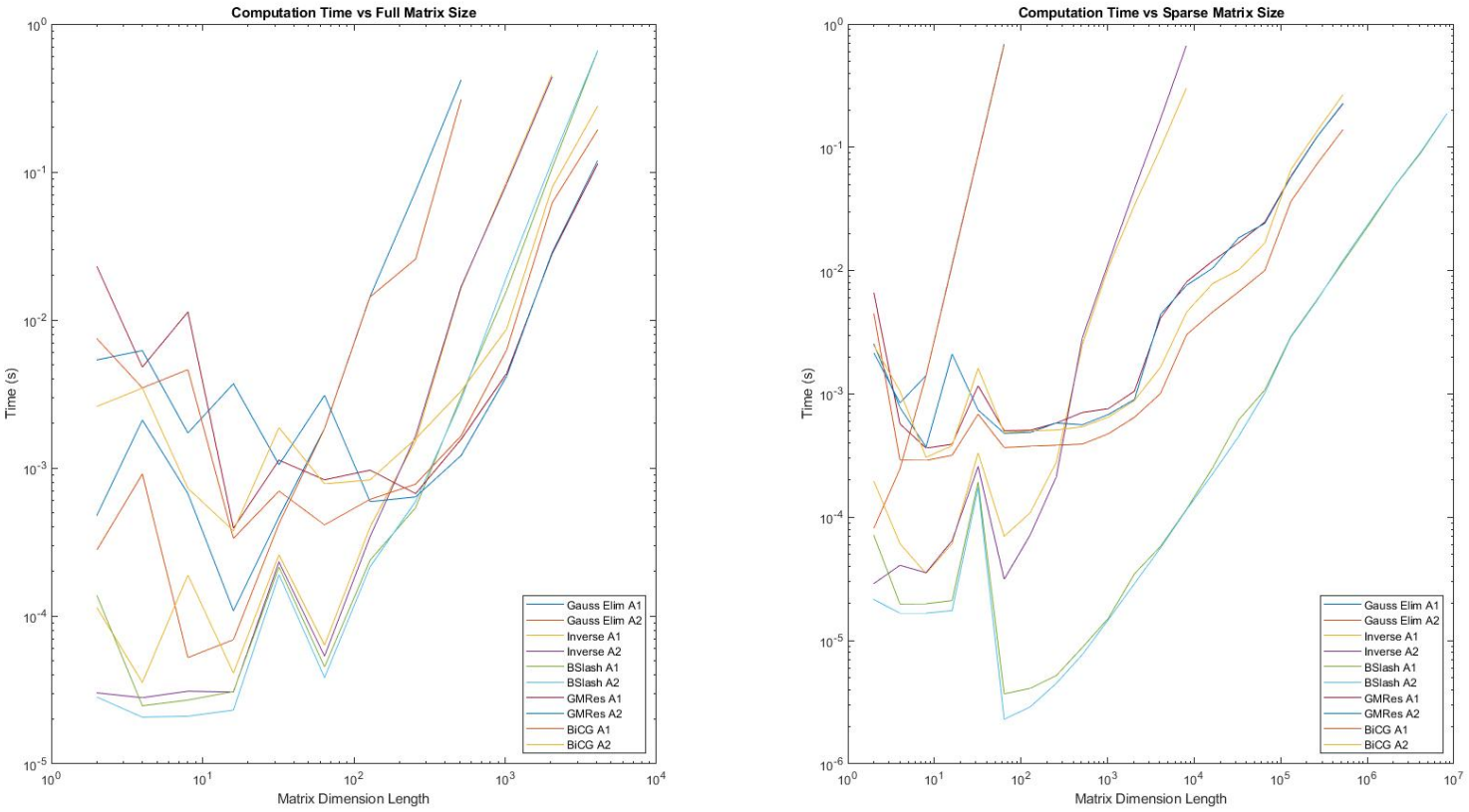

Figure 1: Comparison of Comp. Time vs Matrix Dimensions for Full vs Sparse Matrices

As can be seen, the computation time for solving the linear systems of equations is vastly smaller for practically all methods involving sparse matrices, except for Gaussian Elimination where it is comparable. On observation of Figure 1, it was deduced that for any matrix with side length of about 3×10^2, it would begin to be noticeably faster if used with sparse matrices. The backslash method also appears to be vastly faster than other methods for sparse matrices (at least for the simple coefficient matrices analysed). As a result of this, iterative methods aren't necessarily more beneficial (in terms of execution time). This, however, is only on a small sample size of 2 different matrix types which were banded.
The slope of the lines in each case represent the computation time in "seconds per row/column". As this is a log-log graph, in reality the slope would be much steeper.

Appendices

Appendix 1: R3.1 Matlab Code

```
format long
%PART A

iA = [2, -2, -1;
      4, 1, -2;
      -2, 1, -1];
iiA = [1, 2, -1;
       0, 3, 1;
       2, -1, 1];
iiiA = [2, 1, -4;
        1, -1, 1;
        -1, 3, -2];
ib = [-2; 1; -3];
iib = [2; 4; 2];
iiib = [-7; -2; 6];

i = gausselim(iA, ib);
ii = gausselim(iiA, iib);
iii = gausselim(iiiA, iiib);

%PART B
b1 = gausselim(hilb(2), ones(2, 1));
b2 = gausselim(hilb(5), ones(5, 1));
b3 = gausselim(hilb(10), ones(10, 1));

function x = gausselim(A,b)
    %Gaussian Elimination
    %Input: matrices A and b, where A are the multiples of the unknowns,
    %and b the solutions
    %Output: Matrix x of solutions
    %Source: Huskie, K, Jan, (1)
    [row, ~] = size(A);
    n = row;
    x = zeros(size(b));
    for k = 1:n-1
      for i = k+1:n
         xMultiplier = A(i,k) / A(k,k);
         for j=k+1:n
            A(i,j) = A(i,j) - xMultiplier * A(k,j);
         end
```

```
        b(i, :) = b(i, :) - xMultiplier * b(k, :);
      end
      % backsubstitution:
      x(n, :) = b(n, :) / A(n,n);
      for i = n-1:-1:1
          summation = b(i, :);
          for j = i+1:n
              summation = summation - A(i,j) * x(j, :);
          end
          x(i, :) = summation / A(i,i);
      end
    end
end
```

Appendix 2 - R3.2 Matlab Code

Appendix 2.1 - R3.2a Code

```
n = 10;
A = zeros(n);
x = ones(n, 1);
for j = 1:n
    for i = 1:n
        A(i, j) = double(5 / (i + 2*j - 1));
    end
end
Cond_no = cond(A)
b = A * x;
xc = A\b;
for_err = max(abs(x - xc))
err_mag_fac = (for_err / max(abs(x))) / eps
```

Appendix 2.2 - R3.2b Code

```
n = 500;
A = zeros(n);
x = ones(n, 1);
for j = 1:n
    for i = 1:n
        A(i, j) = abs(i - j) + 1;
    end
end
Cond_no = cond(A)
b = A * x;
xc = A\b;
for_err = max(abs(x - xc))
err_mag_fac = (for_err / max(abs(x))) / eps
```

Appendix 3 - R3.4 Matlab Code

```
clear all

%Following code finds time for full/sparse matrices
for method = 1:10
    if method == 1 %full gaussian elim
        pwer = 1:9;                                          %choose how many powers to
            calculate to
        sizes1 = 2.^pwer;                                    %create array of numbers to
            iterate matrix dimensions over
        [GaussTimeA1, GaussTimeA2] = deal(zeros(size(sizes1)));  %initiate time arrays
        for n = 1:length(sizes1)
            [sA1, b1] = sparsesetup(sizes1(n), 2, 1, 2, [-1, 3, -1]); %create matrices with
                dimensions n*n
            [sA2, b2] = sparsesetup(sizes1(n), 1, 0, -1, [1, 2, 1]);
            A1 = full(sA1); A2 = full(sA2);                  %convert to full matrices
            tic; gausselim(A1, b1); GaussTimeA1(n) = toc;    %time the solving computation,
                store time taken
            tic; gausselim(A2, b1); GaussTimeA2(n) = toc;
        end
    elseif method == 2 %sparse gaussian elim
        pwer = 1:6;
        sizes2 = 2.^pwer;
        [sGaussTimeA1, sGaussTimeA2] = deal(zeros(size(sizes2)));
        for n = 1:length(sizes2)
            [sA1, b1] = sparsesetup(sizes2(n), 2, 1, 2, [-1, 3, -1]);
            [sA2, b2] = sparsesetup(sizes2(n), 1, 0, -1, [1, 2, 1]);
            tic; gausselim(sA1, b1); sGaussTimeA1(n) = toc;
            tic; gausselim(sA2, b1); sGaussTimeA2(n) = toc;
        end
    elseif method == 3 %full inverse func
        pwer = 1:11;
        sizes3 = 2.^pwer;
        [InvTimeA1, InvTimeA2] = deal(zeros(size(sizes3)));
        for n = 1:length(sizes3)
            [sA1, b1] = sparsesetup(sizes3(n), 2, 1, 2, [-1, 3, -1]);
            [sA2, b2] = sparsesetup(sizes3(n), 1, 0, -1, [1, 2, 1]);
            A1 = full(sA1); A2 = full(sA2);
            tic; inv(A1) * b1; InvTimeA1(n) = toc;
            tic; inv(A2) * b1; InvTimeA2(n) = toc;
        end
    elseif method == 4 %sparse inverse func
        pwer = 1:13;
        sizes4 = 2.^pwer;
        [sInvTimeA1, sInvTimeA2] = deal(zeros(size(sizes4)));
        for n = 1:length(sizes4)
            [sA1, b1] = sparsesetup(sizes4(n), 2, 1, 2, [-1, 3, -1]);
            [sA2, b2] = sparsesetup(sizes4(n), 1, 0, -1, [1, 2, 1]);
            tic; inv(sA1) * b1; sInvTimeA1(n) = toc;
            tic; inv(sA2) * b1; sInvTimeA2(n) = toc;
        end
    elseif method == 5 %full backslash
        pwer = 1:12;
        sizes5 = 2.^pwer;
        [SlashTimeA1, SlashTimeA2] = deal(zeros(size(sizes5)));
```

```
    for n = 1:length(sizes5)
        [sA1, b1] = sparsesetup(sizes5(n), 2, 1, 2, [-1, 3, -1]);
        [sA2, b2] = sparsesetup(sizes5(n), 1, 0, -1, [1, 2, 1]);
        A1 = full(sA1); A2 = full(sA2);
        tic; A1\b1; SlashTimeA1(n) = toc;
        tic; A2\b1; SlashTimeA2(n) = toc;
    end
 elseif method == 6 %sparse backslash
    pwer = 1:23;
    sizes6 = 2.^pwer;
    [sSlashTimeA1, sSlashTimeA2] = deal(zeros(size(sizes6)));
    for n = 1:length(sizes6)
        [sA1, b1] = sparsesetup(sizes6(n), 2, 1, 2, [-1, 3, -1]);
        [sA2, b2] = sparsesetup(sizes6(n), 1, 0, -1, [1, 2, 1]);
        tic; sA1\b1; sSlashTimeA1(n) = toc;
        tic; sA2\b1; sSlashTimeA2(n) = toc;
    end
elseif method == 7 % full GMRes
    pwer = 1:12;
    sizes7 = 2.^pwer;
    [GMResTimeA1, GMResTimeA2] = deal(zeros(size(sizes7)));
    for n = 1:length(sizes7)
        [sA1, b1] = sparsesetup(sizes7(n), 2, 1, 2, [-1, 3, -1]);
        [sA2, b2] = sparsesetup(sizes7(n), 1, 0, -1, [1, 2, 1]);
        A1 = full(sA1); A2 = full(sA2);
        tic; gmres(A1, b1); GMResTimeA1(n) = toc;
        tic; gmres(A2, b2); GMResTimeA2(n) = toc;
    end
elseif method == 8 %sparse GMRes
    pwer = 1:19;
    sizes8 = 2.^pwer;
    [sGMResTimeA1, sGMResTimeA2] = deal(zeros(size(sizes8)));
    for n = 1:length(sizes8)
        [sA1, b1] = sparsesetup(sizes8(n), 2, 1, 2, [-1, 3, -1]);
        [sA2, b2] = sparsesetup(sizes8(n), 1, 0, -1, [1, 2, 1]);
        tic; gmres(sA1, b1); sGMResTimeA1(n) = toc;
        tic; gmres(sA2, b2); sGMResTimeA2(n) = toc;
    end
elseif method == 9 %full BiCG
    pwer = 1:12;
    sizes9 = 2.^pwer;
    [BiCGTimeA1, BiCGTimeA2] = deal(zeros(size(sizes9)));
    for n = 1:length(sizes9)
        [sA1, b1] = sparsesetup(sizes9(n), 2, 1, 2, [-1, 3, -1]);
        [sA2, b2] = sparsesetup(sizes9(n), 1, 0, -1, [1, 2, 1]);
        A1 = full(sA1); A2 = full(sA2);
        tic; bicg(A1, b1); BiCGTimeA1(n) = toc;
        tic; bicg(A2, b2); BiCGTimeA2(n) = toc;
    end
elseif method == 10 %sparse BiCG
    pwer = 1:19;
    sizes0 = 2.^pwer;
    [sBiCGTimeA1, sBiCGTimeA2] = deal(zeros(size(sizes0)));
    for n = 1:length(sizes0)
        [sA1, b1] = sparsesetup(sizes0(n), 2, 1, 2, [-1, 3, -1]);
        [sA2, b2] = sparsesetup(sizes0(n), 1, 0, -1, [1, 2, 1]);
```

```
            tic; bicg(sA1, b1); sBiCGTimeA1(n) = toc;
            tic; bicg(sA2, b2); sBiCGTimeA2(n) = toc;
        end
    end
end

%following code produces side-by-side plot of comp time vs matrix
%dimensions
figure
subplot(1, 2, 1)      %subplot for full matrices
loglog(sizes1, GaussTimeA1);
hold on
loglog(sizes1, GaussTimeA2);
loglog(sizes3, InvTimeA1);
loglog(sizes3, InvTimeA2);
loglog(sizes5, SlashTimeA1);
loglog(sizes5, SlashTimeA2);
loglog(sizes7, GMResTimeA1);
loglog(sizes7, GMResTimeA2);
loglog(sizes9, BiCGTimeA1);
loglog(sizes9, BiCGTimeA2);
legend('Gauss Elim A1', 'Gauss Elim A2', 'Inverse A1', 'Inverse A2', 'BSlash A1', ...
    'BSlash A2', 'GMRes A1', 'GMRes A2', 'BiCG A1', 'BiCG A2', 'Location', 'southeast');
title('Computation Time vs Full Matrix Size');
ylabel('Time (s)');
xlabel('Matrix Dimension Length');

subplot(1, 2, 2)      %subplot for sparse matrices
loglog(sizes2, sGaussTimeA1);
hold on
loglog(sizes2, sGaussTimeA2);
loglog(sizes4, sInvTimeA1);
loglog(sizes4, sInvTimeA2);
loglog(sizes6, sSlashTimeA1);
loglog(sizes6, sSlashTimeA2);
loglog(sizes8, sGMResTimeA1);
loglog(sizes8, sGMResTimeA2);
loglog(sizes0, sBiCGTimeA1);
loglog(sizes0, sBiCGTimeA2);
legend('Gauss Elim A1', 'Gauss Elim A2', 'Inverse A1', 'Inverse A2', 'BSlash A1', ...
    'BSlash A2', 'GMRes A1', 'GMRes A2', 'BiCG A1', 'BiCG A2', 'Location', 'southeast');
title('Computation Time vs Sparse Matrix Size');
ylabel('Time (s)');
xlabel('Matrix Dimension Length');

function x = gausselim(A, b)
    %Gaussian Elimination
    %Input: matrices A and b, where A are the multiples of the unknowns,
    %and b the solutions
    %Output: Matrix x of solutions
    %Source: Huskie, K, Jan, (1)
    [row, ~] = size(A);
    n = row;
    x = zeros(size(b));
    for k = 1:n-1
      for i = k+1:n
```

```
        xMultiplier = A(i,k) / A(k,k);
        for j=k+1:n
           A(i,j) = A(i,j) - xMultiplier * A(k,j);
        end
        b(i, :) = b(i, :) - xMultiplier * b(k, :);
      end
        % backsubstitution:
        x(n, :) = b(n, :) / A(n,n);
        for i = n-1:-1:1
            summation = b(i, :);
            for j = i+1:n
                summation = summation - A(i,j) * x(j, :);
            end
            x(i, :) = summation / A(i,i);
        end
    end
end

%following function is the textbook given program :(
function [a, b] = sparsesetup(n, b1, bn, bl, val)
% Input: n = size of system
%        b1, bn, bl = initial b value, filling b values, final b value respectively
%        val = 1x3 array of values for band
    e = ones(n, 1); n2 = n/2;
    a = spdiags([val(1)*e val(2)*e val(3)*e], -1:1, n, n); %entries of a
    a(n2 + 1, n2) = -1; a(n2, n2 + 1) = -1; %fix up to 2 entries
    b = zeros(n, 1);
    b(1) = 2; b(n) = bl; b(2:n-1) = bn;
end
```

References

[1] Huskie, K, Jan, 2018. *Matlab Function: Solving Ax=B using Gaussian Elimination where b is a n x m matrix not necessarily a n x 1 matrix.* Matlab Answers, Accessed 19/09/2020. https://au.mathworks.com/matlabcentral/answers/404921-matlab-function-solving-ax-b-using-gaussian-elimination-where-b-is-a-n-x-m-matrix-not-necessarily-a