MATH4602 Mini Project

Nip Hok Leung, UID: 3035957240

April 20, 2023

a) i) Let $d \geq 2$ and $\mathbf{x} = [x_1, x_2, ...x_{n^2}]^T \neq \mathbf{0}$. Then we have

$$\begin{aligned}\mathbf{x}^T A\mathbf{x} &= 2^d x_1^2 + 2^d x_2^2 + ... + 2^d x_n^2 + \sum_{i=1}^{n^2}\sum_{j=i+1}^{n^2} 2A_{ij}x_i x_j \\ &\geq 4(|x_1|^2 + |x_2|^2 + ... + |x_n|^2) - \sum_{i=1}^{n^2}\sum_{j=i+1}^{n^2} 2|A_{ij}||x_i||x_j| \\ &\geq 2x_1^2 + \sum_{i=1}^{n^2}\sum_{j=i+1}^{n^2} |A_{ij}|(|x_i| - |x_j|)^2\end{aligned}$$

with the last inequality arising from the fact that for each row i, $\sum_{j\neq i}|A_{ij}| \leq 4$, and in particular $\sum_{j=2}^{n^2} 2|A_{1j}| = 4$.

Since we have $A_{1,2} = A_{2,3} = ... = A_{n^2-1,n^2} = -1$, $\sum_{i=1}^{n^2}\sum_{j=i+1}^{n^2} 2|A_{ij}|(|x_i| - |x_j|)^2 = 0$ iff $x_1 = x_2 = ...x_{n^2}$. However, if $x_1 = x_2 = ...x_{n^2}$ and $\mathbf{x} \neq 0$, we must have $x_1^2 > 0$.

Hence $2x_1^2 + \sum_{i\neq j}|A_{ij}|(|x_i| - |x_j|)^2 > 0$, so A is positive definite.

ii) The code is as follows:

```
function numIters = MiniProjectGaussSeidel(n, d, epsilon)
    numIters = 0;
    x = zeros(n^2,1);
    temp = zeros(n^2,1);
    % this b is valid only for n >= 3
    b = zeros(n^2,1);
    for i = 1:n^2
        b(i) = 2^d - 4 + (mod(i,n) == 1 || mod(i,n) == 0) + (i<=n || i > n^2 - n);
    end
    while norm(x-ones(n^2,1),2) > epsilon
        % computing b-U*x and saving it as x
        for i = 1:n^2
            if i == n^2
                temp(i) = b(i);
            elseif i > n^2 - n
                temp(i) = b(i) + x(i+1);
            else
                temp(i) = b(i) + x(i+1)*(mod(i,n)~=0) + x(i+n);
            end
        end
        x = temp;
        % computing L \ x
        for i = 1:n^2
            if i == 1
                temp(i) = x(i)/2^d;
            elseif i <= n
                temp(i) = (x(i)+temp(i-1))/2^d;
            else
                temp(i) = (x(i)+temp(i-1)*(mod(i,n)~=1)+temp(i-n))/2^d;
            end
        end
        x = temp;
        numIters = numIters + 1;
    end
end
```

iii) In each iteration, computing $\mathbf{b} - U\mathbf{x}$ and $L^{-1}(\mathbf{b} - U\mathbf{x})$ both take $O(n^2)$, as each iteration in the loop only involves a constant number of operations. Similarly, computing the error also takes $O(n^2)$ time for the same reason. Hence each iteration has $O(n^2)$ time complexity.

iv) The number of iterations for each pair of values of n, d is shown in the table below:

	$d = 2$	$d = 3$	$d = 4$	$d = 5$
$n = 10$	195	14	8	6
$n = 20$	742	15	9	7
$n = 30$	1657	16	9	7
$n = 40$	2948	16	9	7

The number of iterations needed to converge appears to be decreasing as d increases, and in particular it is very large when $d = 2$, especially for larger values of n.

b) i) The code is as follows:

```
function numIters = MiniProjectBlockJacobi(n, d, epsilon)
    numIters = 0;
    x = zeros(n^2,1);
    temp = zeros(n^2,1);
    % cholesky factorization of A_n
    ldiag = zeros(n,1); % diagonal elements
    llow = zeros(n,1); % elements below the diagonal (starts with dummy 0)
    ldiag(1) = sqrt(2^d);
    for i = 2:n
        llow(i) = -1/ldiag(i-1);
        ldiag(i) = sqrt(2^d-llow(i)^2);
    end
    % this b is valid only for n >= 3
    b = zeros(n^2,1);
    for i = 1:n^2
        b(i) = 2^d - 4 + (mod(i,n) == 1 || mod(i,n) == 0) + (i<=n || i > n^2 - n);
    end
    while norm(x-ones(n^2,1),2) > epsilon
        % computing b-(A-D)*x and storing it in x
        for i = 1:n^2
            if i <= n
                temp(i) = b(i) + x(i+n);
            elseif i > n^2 - n
                temp(i) = b(i) + x(i-n);
            else
                temp(i) = b(i) + x(i-n) + x(i+n);
            end
        end
        x = temp;
        % computing D \ x using the cholesky factorization computed earlier
        % forward substitution
        for i = 1:n
            temp(i*n-n+1) = x(i*n+1-n)/ldiag(1);
            for j = 2:n
                temp(i*n+j-n) = (x(i*n+j-n)-temp(i*n+j-n-1)*llow(j))/ldiag(j);
            end
        end
        x = temp;
        % backward substitution
        for i = 1:n
            temp(i*n) = x(i*n)/ldiag(n);
            for j = n-1:-1:1
                temp(i*n+j-n) = (x(i*n+j-n)-temp(i*n+j-n+1)*llow(j+1))/ldiag(j);
            end
        end
        x = temp;
        numIters = numIters + 1;
    end
end
```

ii) Computing the ith element of $b-(A-D)*x$ involves only a constant number of operations regardless of n. Also, each of the forwards and backwards substitutions only require a constant number of operations to compute the ith element of $D^{-1}(b-(A-D)*x)$ due to the tridiagonal structure of A_n. Hence the time complexity is $O(n^2)$.

iii) The number of iterations for each pair of values of n, d is shown in the table below:

	$d = 2$	$d = 3$	$d = 4$	$d = 5$
$n = 10$	198	14	9	6
$n = 20$	746	15	9	7
$n = 30$	1661	16	9	7
$n = 40$	2952	16	9	7

The number of iterations needed to converge appears to be very similar to the Gauss-Seidel method in part a). It follows the same pattern: decreasing as d increases, and very large when $d = 2$ particularly for larger values of n.

c) i) The code is as follows:

```
function numIters = MiniProjectBlockGaussSeidel(n, d, epsilon)
    numIters = 0;
    x = zeros(n^2,1);
    temp = zeros(n^2,1);
    % cholesky factorization of A_n
    ldiag = zeros(n,1); % diagonal elements
    llow = zeros(n,1); % elements below the diagonal (starts with dummy 0)
    ldiag(1) = sqrt(2^d);
    for i = 2:n
        llow(i) = -1/ldiag(i-1);
        ldiag(i) = sqrt(2^d-llow(i)^2);
    end
    % this b is valid only for n >= 3
    b = zeros(n^2,1);
    for i = 1:n^2
        b(i) = 2^d - 4 + (mod(i,n) == 1 || mod(i,n) == 0) + (i<=n || i > n^2 - n);
    end
    while norm(x-ones(n^2,1),2) > epsilon
        % computing b-(A-L)*x and storing it in x
        for i = 1:n^2
            if i > n^2 - n
                temp(i) = b(i);
            else
                temp(i) = b(i) + x(i+n);
            end
        end
        x = temp;
        % computing L \ x using the cholesky factorization computed earlier
        for i = 1:n
            if (i > 1)
                x(i*n+1-n:i*n) = x(i*n+1-n:i*n) + temp(i*n+1-2*n:i*n-n);
            end
            % forward substitution
            temp(i*n-n+1) = x(i*n+1-n)/ldiag(1);
            for j = 2:n
                temp(i*n+j-n) = (x(i*n+j-n)-temp(i*n+j-n-1)*llow(j))/ldiag(j);
            end
            % backward substitution
            x(i*n+1-n:i*n) = temp(i*n+1-n:i*n);
            temp(i*n) = x(i*n)/ldiag(n);
            for j = n-1:-1:1
                temp(i*n+j-n) = (x(i*n+j-n)-temp(i*n+j-n+1)*llow(j+1))/ldiag(j);
            end
        end
        x = temp;
        numIters = numIters + 1;
    end
end
```

ii) Computing the ith element of $b-(A-L)*x$ involves only a constant number of operations regardless of n. Also, each of the forwards and backwards substitution only require a constant number of operations are to compute the ith element of $D^{-1}(b-(A-D)*x)$ due to the tridiagonal structure of A_n. Hence the time complexity is $O(n^2)$.

iii) The number of iterations for each pair of values of n, d is shown in the table below:

	$d=2$	$d=3$	$d=4$	$d=5$
$n=10$	100	10	7	5
$n=20$	374	11	7	5
$n=30$	831	11	7	6
$n=40$	1477	11	7	6

The number of iterations needed to converge appears to be somewhat lower than the Gauss-Seidel method and block Jacobi method in parts a) and b). Similar to the Gauss-Seidel and block Jacobi method, the number of iterations needed is still large when $d = 2$ especially for larger values of n, however it is approximately half that of the other methods.