Lista 1

Introdução à Análise Numérica

Métodos iterativos para sistemas de equações lineares

Lucas Emanuel Resck Domingues

Setembro de 2020

1. (a) Vamos mostrar que o método converge. Basta que os autovalores sejam menores do que 1. Seja C tal que $x^{(n+1)} = Cx^{(n)} + b$. Então, se v é autovetor e λ é autovalor comrrespondente, segue que:

$$\begin{aligned}(D + wL)^{-1}((1-w)D - wU)v &= \lambda v \\ ((1-w)D - wU)v &= (D + wL)\lambda v\end{aligned}$$

Escolhemos uma linha i. Se $A_{ij} = a_{ij}$, vemos que:

$$\begin{aligned}(1-w)a_{ii}v_i - w\sum_{j=i+1}^{m^2} a_{ij}v_j &= a_{ii}\lambda v_i + \lambda w\sum_{j=1}^{i-1} a_{ij}v_j \\ -a_{ii}\lambda v_i &= \lambda w\sum_{j=1}^{i-1} a_{ij}v_j - (1-w)a_{ii}v_i + w\sum_{j=i+1}^{m^2} a_{ij}v_j \\ |a_{ii}\lambda v_i| &= \left|\lambda w\sum_{j=1}^{i-1} a_{ij}v_j - (1-w)a_{ii}v_i + w\sum_{j=i+1}^{m^2} a_{ij}v_j\right| \\ |a_{ii}\lambda v_i| &= \left|\lambda w\sum_{j=1}^{i-1}(-a_{ij})v_j + (1-w)a_{ii}v_i + w\sum_{j=i+1}^{m^2}(-a_{ij})v_j\right|\end{aligned}$$

Suponhamos que as coordenadas de v não sejam todas iguais. Segue que existe k tal que v_k é estritamente maior do que algum v_i. Então

é fácil ver que:

$$
\begin{aligned}
|a_{ii}\lambda| &= \frac{1}{|v_i|}\left|\lambda w\sum_{j=1}^{i-1}(-a_{ij})v_j + (1-w)a_{ii}v_i + w\sum_{j=i+1}^{m^2}(-a_{ij})v_j\right| \\
&< \left|\lambda w\sum_{j=1}^{i-1}(-a_{ij}) + (1-w)a_{ii} + w\sum_{j=i+1}^{m^2}(-a_{ij})\right| \\
|4\lambda| &< |2\lambda w + (1-w)4 + 2w|
\end{aligned}
$$

Caso nossa suposição não seja válida, ou seja, todos os componentes de v são iguais, então escolhemos a primeira linha de A que possui a soma da linha (com exceção da diagonal), em módulo, menor do que 4. Então vale que:

$$
\begin{aligned}
|a_{ii}\lambda| &= \left|\lambda w\sum_{j=1}^{i-1}(-a_{ij}) + (1-w)a_{ii} + w\sum_{j=i+1}^{m^2}(-a_{ij})\right| \\
&\leq |\lambda|\left|w\sum_{j=1}^{i-1}(-a_{ij})\right| + \left|(1-w)a_{ii} + w\sum_{j=i+1}^{m^2}(-a_{ij})\right| \\
&< |\lambda|\,|2w| + |(1-w)4 + w2|
\end{aligned}
$$

Em ambos os casos, temos

$$
\begin{aligned}
|\lambda| &< |\lambda|\left|\frac{w}{2}\right| + \left|1-\frac{w}{2}\right| \\
|\lambda| &< \frac{\left|1-\frac{w}{2}\right|}{\left|1-\frac{w}{2}\right|} \\
&= 1
\end{aligned}
$$

Então o método SOR converge para essa matriz, desde que $0 < w < 2$, para qualquer m natural.

(b) O código do método SOR para essa matriz implementado em Matlab pode ser conferido no Apêndice A.

A matriz em si não foi implementada, pois seus valores são previsíveis no momento da iteração. Isso é bom, pois guardar matrizes tão grandes na memória é muito custoso. No caso de uma matriz esparsa, é custoso criar a matriz. Para a previsão de qual o valor dos termos da matriz durante a iteração, fez-se necessário o cálculo do resto de uma divisão (função mod), que, ao invés de ser calculada em toda iteração, foi calculada para todos os valores e salvas em um vetor

antes da iteração. Por mais que utilizamos uma memória que pode ser substancial no caso de matrizes muito grandes, neste exercício, para os tamanhos de matrizes utilizadas, o custo-benefício se justificou.

Verifiquei, através de alguns experimentos, que o algoritmo não é muito sensível à variação no valor inicial do vetor x ($x^{(0)}$). Dessa forma, um vetor inicial razoável é $x^{(0)} = (0, \cdots, 0)$.

Os valores de n para diferentes valores de m e de ω, com tolerância de 10^{-6}, podem ser conferidos na Tabela 1.

		ω						
		1	**0.5**	**0**	**1.8840***	**1.9397***	**1.9937***	**1.9987***
m	**50**	3950	11855	X	161	-	-	-
	100	15494	46488	X	-	318	-	-
	1000	>>	>>	X	-	-	3157	-
	5000	>>	>>	X	-	-	-	15770

Tabela 1: Valores de n para diversos valores de m e de ω para o método SOR. Os valores de ω com asterisco ($*$) significam os valores ótimos dos respectivos valores de m. X significa que o algoritmo não convergiu. >> significa que o algoritmo leva muito tempo (não necessariamente muitas iterações) para ser realizado com os recursos computacionais disponíveis para este trabalho.

Fica óbvio que os valores de ω ótimos para cada n performaram muito melhor, para cada m. Na verdade, para alguns valores de m, como 1000 e 5000, ficou quase impossível realizar os cálculos com meus recursos computacionais para outros valores de ω, por mais que a teoria garanta a convergência.

Sabemos que, se o algoritmo converge, então $0<\omega<2$. Então, para $\omega = 0$, o método não converge, o que foi verificado com a implementação.

2. (a) O código em Matlab para a função solicitada se encontra no Apêndice B.

Verifica-se de início se n é par e maior do que ou igual a 4, para que faça sentido com a definição da matriz no enunciado do exercício. Os dois algoritmos, Jacobi e Seidel, são realizados de forma concomitante, no mesmo *for loop* de iteração do vetor x. Para isso, basta que as atualizações do Jacobi, ao iterar o vetor x, não influenciem as atualizações dos outros componentes de x, até que se termine a iteração do vetor. Para isso, ao iniciar a iteração sobre cada vetor x^k, salva-se um vetor x_{old} para o Jacobi.

A matriz A e o vetor b não foram implementados. Os valores de a_{ij} e b_i são previsíveis no momento da iteração, através dos valores de i.

Por mais que os dois algoritmos sejam concomitantes nos mesmos *loops*, acontece de que eles podem parar em momentos diferentes, por meio de declarações *if*.

(b) Para $n = 100000$ e $\varepsilon = 10^{-6}$, obtive $k_{jacobi} = 45$ e $k_{seidel} = 37$.

(c) O código em Matlab para a função solicitada se encontra no Apêndice C.

Verifica-se se n é par e maior do que ou igual a 4. Inicia-se $x^{(0)} = (0, \cdots, 0)$. Inicia-se os outros vetores relevantes. O método implementado é aquele visto em aula. Porém, ele foi modificado para que sirva para essa matriz A de forma mais eficiente. A multiplicação da matriz (que é naturalmente esparsa) pelos vetores correspondentes é feita de forma elemento a elemento, de modo análogo ao que foi feito no exercício 2.(a).

O método foi executado. Para $k_{jacobi} = 45$, obtivemos erro de $4.4409 \cdot 10^{-16}$, enquanto que, para $k_{seidel} = 37$, o erro foi o mesmo $4.4409 \cdot 10^{-16}$. Ora, ambos os métodos de Jacobi e Seidel possuem esse valor de k para uma tolerância de 10^{-6}. Então concluímos que o método do gradiente conjugado teve melhor performance no mesmo número de iterações.

A Método SOR

```
function b = b_vector(m)
    b = sparse(m^2, 1);
    b(1:m) = 1;
    b((m^2-m+1):m^2) = 1;
    for i=1:m
        b(i*m) = b(i*m) + 1;
        b(i*m-m+1) = b(i*m-m+1) + 1;
    end
    b = b*2;
end

function [x, n, time] = sor(w, m)
    tic
    % Create the b vector
    b = full(b_vector(m));
    m2 = m^2;
    x = zeros(m2, 1);
    err = 1;
    n = 0;
    % Compute the mods for latter use (many times used)
    mod_vector = zeros(m2, 1);
    for i=1:m2
        mod_vector(i) = mod(i, m);
    end
    while err > 10^-6
```

```
        for i=1:m2
            sum = 0;
            if i > m
                sum = sum - x(i-m);
            end
            if i <= m2-m
                sum = sum - x(i+m);
            end
            mod_i = mod_vector(i);
            if mod_i ~= 0
                sum = sum - x(i+1);
            end
            if mod_i ~= 1
                sum = sum - x(i-1);
            end
            % More efficient formula
            % Only one multiplication
            x(i) = x(i) + w*((b(i)-sum)/4-x(i));
        end
        err = norm(x-2, 'inf');
        n = n + 1;
    end
    time = toc;
end
```

B Métodos de Jacobi e Seidel

```
function [x_jacobi, x_seidel, k_jacobi, k_seidel] = jacobi_seidel(n, epsilon)
    tic
    if n < 4
        error('n is less than 4.');
    else
        if mod(n, 2) ~= 0
            error('n is not an ever number.');
        end
    end
    x_jacobi = zeros(n, 1);
    x_seidel = zeros(n, 1);
    err_jacobi = 1;
    err_seidel = 1;
    k_jacobi = 0;
    k_seidel = 0;
    while err_jacobi >= epsilon || err_seidel >= epsilon
        % Jacobi needs the old values when iterating
        x_jacobi_old = x_jacobi;
        for i=1:n
```

```
% Here we only continue to iterate Seidel if
% it's necessary
if err_seidel >= epsilon
    sum_s = 0;
    if i == n/2 || i == n/2+1
        sum_s = sum_s + x_seidel(i-1);
        sum_s = sum_s + x_seidel(i+1) + 1;
        sum_s = sum_s/3;
    else
        if i == 1
            sum_s = sum_s + x_seidel(2);
            sum_s = sum_s - 1/2*x_seidel(n) + 5/2;
            sum_s = sum_s/3;
        else
            if i == n
                sum_s = sum_s + x_seidel(n-1);
                sum_s = sum_s - 1/2*x_seidel(1) + 5/2;
                sum_s = sum_s/3;
            else
                sum_s = sum_s + x_seidel(i-1);
                sum_s = sum_s + x_seidel(i+1);
                sum_s = sum_s - 1/2*x_seidel(n+1-i) + 3/2;
                sum_s = sum_s/3;
            end
        end
    end
    x_seidel(i) = sum_s;
end
if err_jacobi >= epsilon
    sum_j= 0;
    if i == n/2 || i == n/2+1
        sum_j = sum_j + x_jacobi_old(i-1);
        sum_j = sum_j + x_jacobi_old(i+1) + 1;
        sum_j = sum_j/3;
    else
        if i == 1
            sum_j = sum_j + x_jacobi_old(2);
            sum_j = sum_j - 1/2*x_jacobi_old(n) + 5/2;
            sum_j = sum_j/3;
        else
            if i == n
                sum_j = sum_j + x_jacobi_old(n-1);
                sum_j = sum_j - 1/2*x_jacobi_old(1) + 5/2;
                sum_j = sum_j/3;
            else
                sum_j = sum_j + x_jacobi_old(i-1);
```

```
                                sum_j = sum_j + x_jacobi_old(i+1);
                                sum_j = sum_j - 1/2*x_jacobi_old(n+1-i) + 3/2;
                                sum_j = sum_j/3;
                            end
                        end
                    end
                    x_jacobi(i) = sum_j;
                end
            end
            % We do not update k nor time nor error for Jacobi if
            % it's not necessary anymore
            if err_jacobi >= epsilon
                err_jacobi = norm(x_jacobi-1,'inf');
                k_jacobi = k_jacobi + 1;
                toc_jacobi = toc;
            end
            % The same for Seidel
            if err_seidel >= epsilon
                err_seidel = norm(x_seidel-1,'inf');
                k_seidel = k_seidel + 1;
                toc_seidel = toc;
            end
        end
        display([toc_jacobi, toc_seidel]);
    end
```

C Método do gradiente conjugado

```
function [err_jacobi, err_seidel, x] = conj_grad(n, k_jacobi, k_seidel)
    % Check if n is OK
    if n < 4
        error('n is less than 4.');
    else
        if mod(n, 2) ~= 0
            error('n is not an ever number.');
        end
    end
    % Start x(0)
    x = zeros(n, 1);
    % Start r_0 = b - Ax(0) = b = (5/2, ...)
    r = zeros(n, 1);
    r(1:n) = 3/2;
    r(1) = 5/2;
    r(n) = 5/2;
    r(n/2) = 1;
    r(n/2+1) = 1;
```

```
% Start d_0 = r_0
d = r;

err_jacobi = 0;
err_seidel = 0;

for i=1:n
    % Stop condition
    if sum(abs(r)) == 0
        break
    end
    % Calculate the denominator of various fractions
    % d_k^T*A*d_k
    mult = zeros(n, 1);
    for j=1:n
        if j == 1
            mult(1) = 3*d(1)- d(2) +1/2*d(n);
        else
            if j == n
                mult(n) = 3*d(n)- d(n-1) +1/2*d(1);
            else
                if j == n/2 || j == n/2 + 1
                    mult(j) = 3*d(j) - d(j-1) - d(j+1);
                else
                    mult(j) = 3*d(j) - d(j-1) - d(j+1) + 1/2*d(n-j+1);
                end
            end
        end
    end
    denom = d'*mult;
    % Calculate alpha_k
    alpha = d'*r/denom;
    x = x + alpha*d;
    r = r - alpha*mult;
    % Calculate the beta coefficient using old
    % calculated numbers and vectors
    beta = r'*mult/denom;
    d = r - beta*d;
    if i == k_jacobi
        err_jacobi = norm(x-1, 'inf');
    end
    if i == k_seidel
        err_seidel= norm(x-1, 'inf');
    end
    if i >= k_jacobi && i >= k_seidel
        break
```

```
            end
        end
    end
```