Relatório

Aula Prática 3

Nome: Felipe Marques Esteves Lamarca
Matrícula: 211708306

Sumário

1 INTRODUÇÃO

Neste relatório, faço a implementação de três funções para calcular autovalores e autovetores de forma iterativa. As duas primeiras são versões do Método da Potência convencional, e a terceira trata do Método da Potência Deslocada com Iteração Inversa. Neste último caso utilizamos a função de eliminação gaussiana, para resolução de sistemas lineares, desenvolvida na Aula Prática 1.

2 MÉTODO DA POTÊNCIA

Apresento, abaixo, as versões das funções do Método da Potência convencional. Ambas foram desenvolvidas utilizando como base o algoritmo fornecido:

2.1 Método da Potência — versão 1

```
function[lambda, x1, k, n_erro] = Metodo_potencia_1(A, x0, epsilon, M)

    k = 0;
    [valor, ind] = max(abs(x0));
    x0 = x0/x0(ind);
    x1 = A*x0;
    n_erro = epsilon + 1;

    while (k <= M && n_erro >= epsilon)
        [valor, ind] = max(abs(x1));
        lambda = x1(ind);
        x1 = x1/lambda;
        n_erro = norm((x1-x0), %inf);
        x0 = x1;
        x1 = A*x0;
        k = k + 1;
    end

endfunction
```

2.2 Método da Potência — versão 2

```
function[lambda, x1, k, n_erro] = Metodo_potencia_2(A, x0, epsilon, M)

    k = 0;
    x0 = x0/norm(x0, 2);
    x1 = A*x0;
    n_erro = epsilon + 1;

```

```
    while (k <= M && n_erro >= epsilon)
        lambda = x1'*x0;

        if (lambda < 0) then
            x1 = -x1;
        end

        x1 = x1/norm(x1, 2);
        n_erro = norm((x1-x0), 2);
        x0 = x1;
        x1 = A*x0;
        k = k+1;
    end

endfunction
```

MÉTODO DA POTÊNCIA DESLOCADA COM ITERAÇÃO INVERSA

O Método da Potência Deslocada com Iteração Inversa implica a resolução de um sistema linear. Desenvolvi um método para isso na Aula Prática 1, mais especificamente a função Gaussian_Elimination_4( ), que recebe uma matriz A, um vetor b e resolve o sistema Ax = b. No entanto, na ocasião em que fiz a função, o método que utilizei para encontrar o maior pivô foi através de um *loop* que verificava elemento a elemento na posição de pivô para realizar uma permutação de linha e dar prosseguimento à eliminação. Veja abaixo:

```
function [x, C, P]=Gaussian_Elimination_4(A, b)
    C=[A,b];
    [n]=size(C,1);
    P = eye(n,n);   // cria uma matriz identidade que servira para apontar as permutacoes
    for j=1:(n-1)
        maior_pivo = abs(C(j,j))
        for i=j+1 : n
            if abs(C(i, j)) > maior_pivo then maior_pivo = C(i, j); // iteramos pelos elementos da matriz ate que seja encontrado um elemento maior que o pivo na posicao adequada
                a=i; // guarda na variavel "a" a linha do maior
            else a=j;
            end
        end
        C([j, a],:)= C([a, j],:);
        P([j, a],:)= P([a, j],:);  // realiza as mesmas permutacoes realizadas em A durante a eliminacao na matriz P, apontando quais permutacoes ocorreram
```

```
15
16
17          for i=(j+1):n
18              C(i,j)=C(i,j)/C(j,j);
19              C(i,j+1:n+1)=C(i,j+1:n+1)-C(i,j)*C(j,j+1:n+1);;
20          end
21      end
22
23      x=zeros(n,1);
24      x(n)=C(n,n+1)/C(n,n);
25      for i=n-1:-1:1
26          x(i)=(C(i,n+1)-C(i,i:n)*x(i:n))/C(i,i);
27      end
28      C=C(1:n,1:n);
29  endfunction
```

Agora faço isso de forma mais simples combinando as funções max( ) e abs( ) do Scilab, que me permitem obter a posição do elemento (em posição adequada para pivô) de maior valor absoluto:

```
1   function [x, C, P]=Gaussian_Elimination_4(A, b)
2
3       C=[A,b];
4       [n]=size(C,1);
5       P = eye(n,n);
6
7       for j=1:(n-1)
8
9           [valor, ind] = max(abs(C([j:n], j)));
10
11          if j ~= ind then
12              c_ind = ind + j - 1;
13              C([j, c_ind],:)= C([c_ind, j],:);
14              P([j, c_ind],:)= P([c_ind, j],:);
15          end
16
17          for i=(j+1):n
18              C(i,j)=C(i,j)/C(j,j);
19              C(i,j+1:n+1)=C(i,j+1:n+1)-C(i,j)*C(j,j+1:n+1);;
20          end
21      end
22
23      x=zeros(n,1);
24      x(n)=C(n,n+1)/C(n,n);
25
26      for i=n-1:-1:1
27          x(i)=(C(i,n+1)-C(i,i:n)*x(i:n))/C(i,i);
28      end
```

```
29
30      C=C(1:n,1:n);
31
32  endfunction
```

Como já comentei, essa função é utilizada em outra: a Potencia_deslocada_inversa( ), apresentada abaixo. Esta função também foi implementada seguindo o algoritmo sugerido.

```
1  function[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0,
       epsilon, alfa, M)
2
3      k=0;
4      x0 = x0/norm(x0, 2);
5      n_erro = epsilon + 1;
6
7      NewA = A - (alfa*eye(A));
8
9      while (k <= M && n_erro >= epsilon)
10         x1 = Gaussian_Elimination_4(NewA, x0);
11         x1 = x1/norm(x1, 2);
12         lambda = x1'*A*x1;
13
14         if (lambda < 0) then
15             x1 = -x1;
16         end
17
18         n_erro = norm((x1-x0), 2);
19         x0 = x1;
20         k = k+1;
21
22     end
23
24    lambda1 = lambda
25
26 endfunction
```

3 TESTES COM MATRIZES

Agora que já vimos a implementação das funções e seus códigos, podemos começar os testes. Por limitações das funções, precisamos garantir que as matrizes utilizadas nos testes tenham autovalores reais e, para isso, utilizaremos (quase) sempre matrizes simétricas. Para evitar que escrevamos matrizes muito grandes manualmente e para possibilitar que executemos testes com matrizes arbitrariamente grandes, vamos gerar matrizes A aleatórias e calcular A^TA para obter um resultado simétrico. Considere o seguinte código genérico, para o qual o valor de n será substituído conforme o tamanho da matriz que desejamos obter:

```
RandomMatrix = fix(rand(n, n, "uniform")*10);
A = RandomMatrix'*RandomMatrix;
```

O vetor x_0 inicial será sempre composto por 1's.

3.1 Parte 1

Primeiro vamos ver, dentre outros aspectos, em que medida as duas versões do Método da Potência se aproximam e se distanciam no que diz respeito ao tempo de execução e ao número de iterações necessárias para convergência.

3.1.1 Matriz 3 x 3

Considere a matriz a seguir:

```
114.    66.    48.
66.     59.    65.
48.     65.    89.
```

Vamos aplicar as duas funções e verificar o comportamento:

```
// Metodo_potencia_1()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_1(A, x0, 0.001, 100)
 lambda  =
   208.17881

 k  =
   5.

 n_erro  =
   0.0004975

-->toc
 ans  =
   0.076295

// Metodo_potencia_2()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_2(A, x0, 0.001, 100)
 lambda  =
   208.12242

 k  =
   4.

 n_erro  =
```

```
   0.0009727

-->toc
 ans  =
   0.0349070
```

Primeiro, claro, vamos ver se o programa acertou o autovalor dominante:

```
-->max(abs(spec(A)))
 ans  =
   208.12268
```

De fato, as funções parecem estar funcionando bem, pelo menos no caso 3 por 3. Chegamos a um resultado bastante razoável. Veja nesse caso, também, que não há diferenças tão expressivas entre as duas abordagens nesse caso: a segunda versão do Método da Potência convergiu mais rápido, mas com uma iteração de vantagem apenas. No que diz respeito ao tempo de execução, ambas as funções executaram o processo bem rapidamente, mas com alguma vantagem para a segunda aplicação.

3.1.2 Matriz 5 x 5

```
// Metodo_potencia_1()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_1(A, x0, 1E-15, 50)
 lambda  =
   598.20593

 k  =
   16.

 n_erro  =
   5.551D-16

-->toc
 ans  =
   0.02527

// Metodo_potencia_2()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_2(A, x0, 1E-15, 50)
 lambda  =
   598.20593
 k  =
   16.

 n_erro  =
```

```
   3.640D-16

-->toc
 ans  =
   0.011169
```

```
-->max(abs(spec(A)))
 ans  =
   598.20593
```

Neste caso definimos um epsilon menor e também diminuímos o número máximo de iterações permitidas. Veja que os dois métodos param no mesmo número de iterações. Além disso, com epsilon pequeno, o autovalor dominante encontrado pelas funções é exatamente igual àquele apontado pela função spec() do Scilab. O tempo de execução foi novamente menor no caso da Metodo_potencia_2( ), que foi executada em menos da metade do tempo da outra função. Mesmo assim, o tempo de execução em ambos os casos é baixo.

3.1.3 Matriz 7 x 7

```
// Metodo_potencia_1()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_1(A, x0, 1E-10, 50)
 lambda  =
   1071.5484

 k  =
   11.

 n_erro  =
   2.229D-11

-->toc
 ans  =
   0.004696

// Metodo_potencia_2()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_2(A, x0, 1E-10, 50)
 lambda  =
   1071.5484

 k  =
   11.

 n_erro  =
```

```
   1.550D-11

-->toc
 ans  =
   0.0028440
```

```
-->max(abs(spec(A)))
 ans  =
   1071.5484
```

Mais uma vez, utilizando epsilon pequeno, as funções encontraram o exato mesmo valor para o autovalor dominante da matriz aleatória gerada. O número de iterações também foi o mesmo, mas o segundo método gerou um erro menor e foi executado em pouco mais da metade do tempo do primeiro (um padrão que é visualizado no caso anterior).

3.1.4 Matriz 10 x 10

```
// Metodo_potencia_1()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_1(A, x0, 0.01, 100)
 lambda  =
   2020.8867

 k  =
   3.

 n_erro  =
   0.0032785

-->toc
 ans  =
   0.010278

// Metodo_potencia_2()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_2(A, x0, 0.01, 100)
 lambda  =
   2016.8338

 k  =
   3.

 n_erro  =
   0.0013639

```

```
-->toc
 ans  =
    0.030571
```

```
-->max(abs(spec(A)))
 ans  =
    2016.8380
```

No caso da matriz 10 por 10 aleatória gerada, alteramos o epsilon para um valor um pouco maior do que os utilizados em casos anteriores para verificar o comportamento. Agora, com uma precisão menor, note que a primeira versão se distancia em mais de 4 unidades do valor encontrado tanto pela função spec() quanto pela função Metodo_potencia_2(). Em outras palavras, o que podemos perceber é que, com um parâmetro de precisão mais baixo, a segunda versão do Método da Potência funciona notavelmente melhor que a primeira. O tempo de execução, nesse caso, foi muito parecido nos dois casos.

3.1.5 **Matriz 50 x 50**

```
// Metodo_potencia_1()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_1(A, x0, 0.00001, 100)
 lambda  =
    50629.645

 k  =
    4.

 n_erro  =
    0.0000017

-->toc
 ans  =
    0.071948

// Metodo_potencia_2()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_2(A, x0, 0.00001, 100)
 lambda  =
    50629.615

 k  =
    4.

 n_erro  =
    0.0000007
```

```
-->toc
 ans  =
   0.075761
```

```
-->max(abs(spec(A)))
 ans  =
   50629.615
```

Agora temos uma matriz ainda maior, 50 por 50. Mantendo uma precisão alta, note que a função Metodo_potencia_2( ) ficou exatamente igual à encontrada pela função spec() do Scilab. A Metodo_potencia_1() também ficou bastante próxima. Ambas mantiveram um erro baixo e o mesmo número de iterações, com tempo de execução também parecido.

3.1.6 **Matriz 100 x 100**

```
// Metodo_potencia_1()
-->[lambda, k, n_erro] = Metodo_potencia_1(A, x0, 1E-15, 50)
 lambda  =
   206042.94

 k  =
   10.

 n_erro  =
   6.661D-16

-->toc
 ans  =
   0.0374570

// Metodo_potencia_2()
-->tic

-->[lambda, k, n_erro] = Metodo_potencia_2(A, x0, 1E-15, 50)
 lambda  =
   206042.94

 k  =
   9.

 n_erro  =
   3.216D-16

-->toc
 ans  =
   0.074929
```

```
-->max(abs(spec(A)))
 ans  =
   206042.94
```

Finalmente, nosso último teste é com uma matriz 100 por 100. Agora, a segunda versão da função que implementa o Método da Potência converge mais rápido, precisando de uma iteração a menos que a primeira versão. Além disso, o autovalor dominante é calculado com precisão praticamente exata nos dois casos, com vantagem, agora, para a primeira versão no que diz respeito ao tempo de execução. Como em todos os outros casos, o erro da segunda versão é menor que o da primeira.

3.2 Parte 2

Vamos testar agora como as funções se comportam quando utilizamos uma matriz com autovalor dominante negativo. Para isso, vamos gerar uma matriz diagonal aleatória (e, portanto, as entradas da diagonal correspondem aos autovalores) de tamanho n por n com todas as entradas negativas:

```
diag(fix(rand(n, 1, "uniform")*10))*(-1) - diag(diag(eye(n,n)))
```

Veja que subtraímos uma matriz identidade dessa matriz diagonal gerada. Isso serve simplesmente para evitar que haja entradas nulas na matriz diagonal que vamos utilizar.

Vamos testar as versões do Método da Potência com a matriz A, gerada aleatoriamente pelo código acima:

```
-5.    0.    0.    0.     0.    0.    0.
 0.   -3.    0.    0.     0.    0.    0.
 0.    0.   -9.    0.     0.    0.    0.
 0.    0.    0.   -10.    0.    0.    0.
 0.    0.    0.    0.    -2.    0.    0.
 0.    0.    0.    0.     0.   -7.    0.
 0.    0.    0.    0.     0.    0.   -1.
```

```
-->Metodo_potencia_1(A, x0, 1E-10, 100)
 ans  =
  -10.
```

```
-->Metodo_potencia_2(A, x0, 1E-10, 100)
 ans  =
  -10.000000
```

Não há problema algum. Isso porque a nossa implementação considera a posição da entrada de maior valor absoluto para encontrar lambda, e não o valor absoluto em si. Nesse caso, não há incorrespondência entre o retorno da função e o valor verdadeiro do autovalor dominante.

4 DISCOS DE GERSCHGORIN E MÉTODO DA POTÊNCIA DESLOCADA COM ITERAÇÃO INVERSA

Vamos utilizar os Discos de Gerschgorin para estimar os autovalores de uma determinada matriz gerada aleatoriamente:

```
149.    91.    32.
91.     89.    10.
32.     10.    17.
```

Pelas propriedades dos Discos de Gerschgorin, sabemos que temos 3 discos. O primeiro está centrado no ponto (149, 0) e possui raio 123; o segundo está centrado em (89, 0) e tem raio 101; e o terceiro está centrado em (17, 0) com raio 42. Veja a disposição das circunferências abaixo:

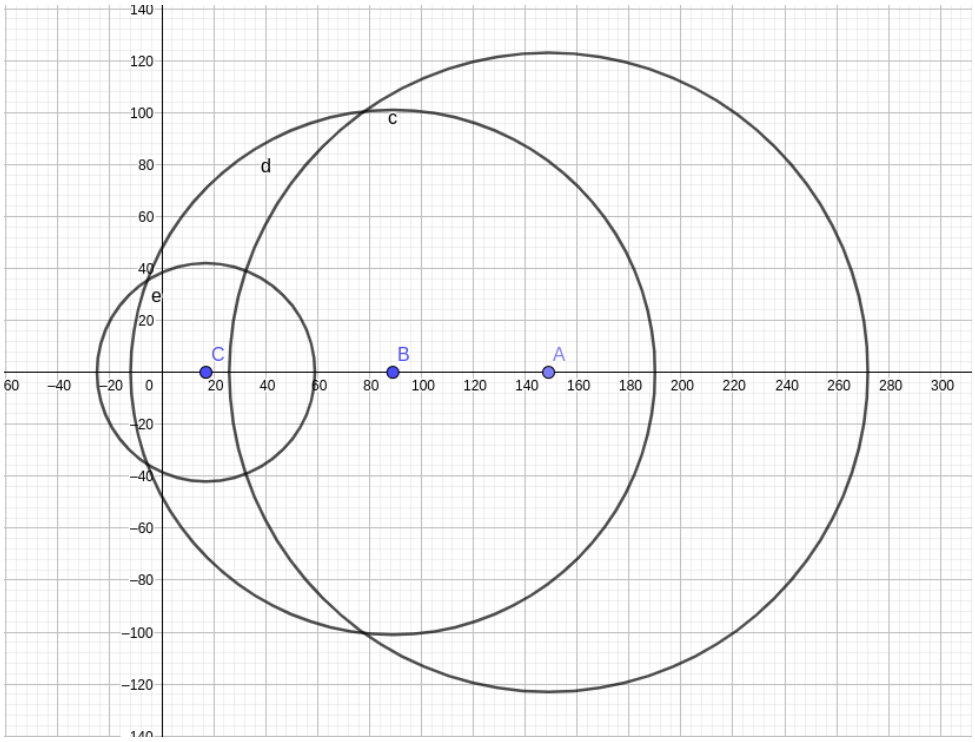

Figura 1 – Discos de Gerschgorin

Antes de testar o funcionamento da nossa função, vamos ver quais são os autovalores dessa matriz para que possamos comparar os resultados:

```
-->spec(A)
 ans   =
   5.6847642
   29.5008
   219.81444
```

Agora vamos buscar os 3 autovalores utilizando 149, 89 e 17 como parâmetros de alfa:

```
-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-15,
   149, 70)
 lambda1  =
   219.81444

 x1  =
   0.8053485
   0.5721041
   0.1552759

 k  =
   65.

 n_erro  =
   6.206D-16
```

Encontramos o primeiro autovalor com bastante precisão, mas note que quase alcançamos o limite de iterações que defini (70). Nos testes com o parâmetro alfa valendo 89 e 17, não conseguimos convergir com essa quantidade de iterações

```
-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-15,
   89, 70)
 lambda1  =
   29.500800

 x1  =
  -0.4314663
   0.7453221
  -0.5082635

 k  =
   71.

 n_erro  =
   2.0000000
```

```
-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-15,
   17, 70)
 lambda1  =
   5.6847653

 x1  =
   0.4066009
  -0.3424896
  -0.8469809

 k  =
   71.
```

```
 n_erro  =
   2.0000000
```

De fato, é verdade que chegamos a valores bastante parecidos aos autovalores calculados pela função spec() do Scilab. No entanto, mesmo aumentando o número de iterações permitidas para 100, 500 e até 1000, ainda não alcançamos convergência:

```
-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-15,
   89, 100)
 lambda1  =
   29.500800

 x1  =
  -0.4314663
   0.7453221
  -0.5082635

 k  =
   101.

 n_erro  =
   2.0000000

-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-15,
   89, 500)
 lambda1  =
   29.500800

 x1  =
  -0.4314663
   0.7453221
  -0.5082635

 k  =
   501.

 n_erro  =
   2.

-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-15,
   89, 1000)
 lambda1  =
   29.500800

 x1  =
  -0.4314663
   0.7453221
```

```
 -0.5082635

k  =
  1001.

n_erro  =
  2.
```

Para fins de teste, podemos simplesmente alterar o valor do parâmetro alfa e ver até que ponto o lambda ainda é calculado no mesmo valor. Veja no exemplo abaixo, por exemplo, em que calculo lambda utilizando 70 e 40 como parâmetros alfa:

```
-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-2,
   70, 1000)
 lambda1  =
   29.500800

 x1  =
  -0.4314663
   0.7453221
  -0.5082635

 k  =
   1001.

 n_erro  =
   2.

-->[lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, 1E-2,
   40, 1000)
 lambda1  =
   29.500800

 x1  =
  -0.4314663
   0.7453221
  -0.5082635

 k  =
   1001.

 n_erro  =
   2.0000000
```

Note que o valor de lambda, nesse caso, continua exatamente o mesmo que aquele calculado para alfa = 89.

5 TESTE COMPLEMENTAR

Desde o início deste relatório estamos testando as funções garantindo que as matrizes de teste possuem todos os autovalores reais. O que fizemos, na prática, foi gerar uma matriz aleatória A e calcular A^TA, já que sabemos que matrizes simétricas dispõem da propriedade desejada. Isso, no entanto, é limitante na medida em que não sabemos o que ocorre caso as matrizes de teste possuam autovalores com parte imaginária. Vamos criar uma matriz aleatória com entradas complexas:

```
-->A = rand(3,3)+%i*rand(3,3)
 A  =
   0.2113249 + 0.068374i    0.3303271 + 0.7263507i   0.8497452 +
   0.2320748i
   0.7560439 + 0.5608486i   0.6653811 + 0.1985144i   0.685731  +
   0.2312237i
   0.0002211 + 0.6623569i   0.6283918 + 0.5442573i   0.8782165 +
   0.2164633i
```

Os autovalores dessa matriz são dados por:

```
-->spec(A)
 ans  =
   1.729703  + 1.1712676i
  -0.164593  - 0.7253954i
   0.1898125 + 0.0374795i
```

Vejamos como as versões das funções que implementam o Método da Potência funcionam com essa matriz:

```
// versao 1
-->[lambda, x0, k, n_erro] = Metodo_potencia_1(A, x0, 1E-15, 100)
 lambda  =
   1.729703 + 1.1712676i

 x0  =
   1.1887322 + 1.2496685i
   1.729703  + 1.1712676i
   1.2414058 + 1.5029717i

 k  =
   34.

 n_erro  =
   5.118D-16

// versao 2
-->[lambda, x0, k, n_erro] = Metodo_potencia_2(A, x0, 1E-15, 100)
at line    11 of function %s_1_s            ( /usr/share/scilab/modules/
   overloading/macros/%s_1_s.sci line 23 )
```

```
in builtin                Metodo_potencia_2 ( /home/felipelmc/ rea  de
   Trabalho/FGV/data-science/3-PERIODO/ALN/Aulas-praticas/Aula03/
   MetodoPotencia.sce line 21 )

Complex comparison not supported. Please define %s_1_s_custom() or check
    your code.
```

Note que, pela implementação utilizada, caso desejássemos utilizar matrizes com entradas complexas para encontrar o autovalor dominante, só poderíamos utilizar a abordagem implementada na primeira versão do Método da Potência. Veja que a função não apenas é executada, como o autovalor dominante é identificado com altíssima precisão. No segundo caso, no entanto, em que tentamos executar a função Metodo_potencia_2(), o código retorna um erro informando que a implementação faz uma comparação com números complexos, o que não é válido na forma como foi definido no código.