Atividade prática 3
Álgebra Linear Numérica

Pedro Henrique Coterli

2 de maio de 2024

Exercício 1

Inicialmente, façamos a função da 1° versão do método da potência, ou seja, normalizando o vetor de modo que sua entrada de maior módulo seja igual a 1.

```
// Função da versão 1 do método da potência
function [lambda, x1, k, n_erro] = Metodo_potencia_v1(A, x0, epsilon, M)
    // Número de iterações
    k = 0
    // Inicializando o erro
    n_erro = epsilon + 1
    // Inicializando x1
    [max_value, imax] = max(abs(x0))
    x1 = x0/x0(imax)

    // Enquanto o número de iterações for suportado e o erro for grande...
    while k < M && n_erro >= epsilon
        // Atualiza os vetores
        x0 = x1
        x1 = A*x1
        // Salva o valor de lambda
        abs_max = max(abs(x0))
        [max_value, imax] = max(abs(x0))
        lambda = x1(imax)
        // Normaliza o vetor
        x1 = x1/lambda
        // Calcula o erro
        n_erro = norm(x1 - x0, "inf")
        // Aumenta uma iteração
        k = k + 1
    end
endfunction
```

Agora, vamos à versão 2, que utiliza o Quociente de Rayleigh.

```
// Função da versão 2 do método da potência
function [lambda, x1, k, n_erro] = Metodo_potencia_v2(A, x0, epsilon, M)
    // Número de iterações
    k = 0
    // Inicializando o erro
    n_erro = epsilon + 1
    // Inicializando x1
    x_norm = norm(x0, 2)
    x1 = x0/x_norm

    // Enquanto o número de iterações for suportado e o erro for grande...
    while k < M && n_erro >= epsilon
        // Atualiza os vetores
        x0 = x1
        x1 = A*x1
        // Salva o valor de lambda de acordo com o Quociente de Rayleigh
        lambda = x1' * x0
        // Se o autovalor for negativo, inverte o vetor
        if lambda < 0
            x1 = -x1
        end
        // Normaliza o vetor
        x_norm = norm(x1, 2)
        x1 = x1/x_norm
        // Calcula o erro
        n_erro = norm(x1 - x0, 2)
        // Aumenta uma iteração
        k = k + 1
    end
endfunction
```

Exercício 2

Antes de fazermos o código para esse método, precisamos de duas funções auxiliares: uma para calcular a decomposição LU de uma matriz e outra para resolver um sistema com essa decomposição. Ambas são adaptações da `Guassian_Elimination_4` da atividade prática 1. Essa separação foi realizada para otimizar o algoritmo, já que não é necessário calcular a decomposição LU de A a cada iteração, mas a solução do sistema sim.

```
// Função para calcular a decomposição LU
function [C, P] = Decomposicao_LU(A)
    C = A
    n = size(C, 1)
    // Inicializando a matriz P
    P = eye(n, n)
    for j = 1:(n - 1)
```

```
        // Se o pivô não for o maior valor de sua coluna, troca as linhas
        if max(abs(C(j:n,j))) ~= abs(C(j,j)) then
            // Encontrando a linha com o maior pivô em módulo
            moduled_vector = abs(C(j:n,j))
            max_pivot_index = find(moduled_vector == max(moduled_vector))
            // Trocando essas linhas na C e na P
            C([j j + max_pivot_index - 1],:) = C([j + max_pivot_index - 1 j],:)
            P([j j + max_pivot_index - 1],:) = P([j + max_pivot_index - 1 j],:)
        end
        // O pivô está na posição (j,j)
        for i = (j + 1):n
            // O elemento C(i, j) é o elemento na posição (i, j) da L na␣
↪decomposição LU de A
            C(i, j) = C(i, j)/C(j, j)
            // Linha i <- Linha i - C(i,j) * Linha j
            // Somente os elementos da diagonal ou acima dela são computados
            // (aqueles que compõem a matriz U)
            C(i, j+1:n)=C(i, j+1:n)-C(i, j)*C(j, j+1:n);
        end
    end
endfunction

// Função para resolver um sistema usando a decomposição LU
function [x] = Resolve_com_LU(C, P, b)
    n = size(C, 1);
    // Inicializando x
    x = zeros(n, 1);
    // Ajustando o sistema de acordo com a matriz de permutação
    b = P*b;
    // Resolvendo Ly = b
    y = zeros(n, 1);
    y(1) = b(1);
    for i = 2:n
        y(i) = b(i) - C(i, 1:i-1) * y(1:i-1);
    end
    // Resolvendo Ux = y
    x(n) = y(n) / C(n, n);
    for i = n-1:-1:1
        x(i) = (y(i) - C(i, i+1:n) * x(i+1:n)) / C(i, i);
    end
endfunction
```

OBS.: Para a função `Resolve_com_LU`, foi necessário resolver os dois sistemas triangulares da equação $LUx = Pb$, em oposição à função original `Gaussian_Elimination_4`, que calculava apenas a segunda parte ($Ux = y$). Isso se deu devido ao fato de que, na função original, as alterações realizadas na matriz para obter a decomposição LU (permutações e operações entre linhas) afetavam ao mesmo tempo o vetor b, que estava "acoplado" à C formando a matriz aumentada $[C, b]$. Assim, ela era capaz de resolver apenas a segunda parte do sistema, pois a resposta da primeira parte já

estava ali. No entanto, com a separação das funções realizada acima, a função `Resolve_com_LU` não consegue ter um acesso prático a essas mudanças realizadas pela função `Decomposicao_LU`, de forma que torna-se necessário resolver manualmente também a primeira fase do sistema ($Ly = b$).

Agora sim, vamos à função do método:

```
// Função do método da potência deslocada com iteração inversa
function [lambda1, x1, k, n_erro] = Potencia_deslocada_inversa(A, x0, epsilon,␣
↪alfa, M)
    // Inicializando o número de iterações
    k = 0
    // Normalizando x0
    x_norm = norm(x0, 2)
    x0 = x0/x_norm
    // Inicializando o erro
    n_erro = epsilon + 1

    // Dimensão de A
    n = size(A, 1)
    // Calculando a decomposição LU de (A - alfa*I)
    [C, P] = Decomposicao_LU(A - alfa * eye(n, n))

    // Enquanto o número de iterações for suportado e o erro for grande...
    while k < M && n_erro >= epsilon
        // Calcula x1 resolvendo o sistema (A - alfa*I)*x1 = x0
        x1 = Resolve_com_LU(C, P, x0)
        // Normaliza x1
        x_norm = norm(x1, 2)
        x1 = x1/x_norm
        // Calcula lambda de acordo com o Quociente de Rayleigh
        lambda1 = x1'*A*x1
        // Se os vetores estiverem em sentidos opostos, inverte o vetor
        if x1'*x0 < 0
            x1 = -x1
        end
        // Calcula o erro
        n_erro = norm(x1 - x0, 2)
        // Atualiza os vetores
        x0 = x1
        k = k + 1
    end
endfunction
```

Exercício 3

Vamos testar ambas as funções do método da potência com algumas matrizes A:

```
A1 = [3/2 -1/2 0; -1/2 3/2 0; 0 0 3]
```

```
A1  =
  1.5  -0.5   0.
 -0.5   1.5   0.
  0.    0.    3.
```

Os autovalores dessa matriz são 1, 2 e 3. Assim, nossas funções devem retornar 3 como autovalor.

```
disp("===== Versão 1 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v1(A1, [1; 2; 3], 10^(-5), 30)
disp("===== Versão 2 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v2(A1, [1; 2; 3], 10^(-5), 30)
```

```
 "===== Versão 1 ====="
lambda  =
  3.
x1  =
 -0.0000149
  0.0000149
  1.
k  =
  23.
n_erro  =
  0.0000074

 "===== Versão 2 ====="
lambda  =
  3.0000000
x1  =
 -0.0000099
  0.0000099
  1.0000000
k  =
  24.
n_erro  =
  0.0000070
```

Ambas retornaram corretamente, com a versão 1 levando uma iteração a menos que a 2. Vamos ao próximo exemplo.

```
A2 = [1/2 -5*sqrt(2)/4 5*sqrt(2)/4 0;
      -5*sqrt(2)/4 -1/4 -3/4 0;
      5*sqrt(2)/4 -3/4 -1/4 0;
      0 0 0 5]
```

```
A2  =
  0.5       -1.767767   1.767767   0.
 -1.767767  -0.25       -0.75       0.
  1.767767  -0.75       -0.25       0.
  0.         0.         0.         5.
```

Seus autovalores são -2, -1, 3 e 5, dando a entender que nossas funções retornarão o valor 5. Vamos conferir isso:

```
disp("===== Versão 1 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v1(A2, [3; 0; 0; 4], 10^(-10), 50)
disp("===== Versão 2 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v2(A2, [3; 0; 0; 4], 10^(-10), 50)
```

```
 "===== Versão 1 ====="
lambda  =
  5.
x1  =
  1.083D-10
 -7.656D-11
  7.656D-11
  1.
k  =
  43.
n_erro  =
  7.218D-11

 "===== Versão 2 ====="
lambda  =
  5.
x1  =
  6.497D-11
 -4.594D-11
  4.594D-11
  1.
k  =
  44.
n_erro  =
  6.125D-11
```

Mais uma vez, a versão 1 obteve a aproximação solicitada com uma iteração a menos que a 2° versão.

Façamos um próximo teste com a seguinte matriz:

```
rand("seed", 2)
A3 = round(10*rand(5, 5))
```

```
A3  =
  10.   2.   4.   10.   1.
  3.    2.   5.   6.    4.
  3.    7.   1.   10.   6.
  6.    6.   5.   6.    4.
  4.    3.   6.   6.    8.
```

Abaixo estão seus autovalores:

```
spec(A3)
```

```
 ans  =
   25.874461 + 0.i
   7.0293042 + 0.i
  -4.4923435 + 0.i
  -3.4373729 + 0.i
   2.0259514 + 0.i
```

Dessa forma, o resultado esperado é o primeiro desses valores. Vejamos:

```
disp("===== Versão 1 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v1(A3, [1; 5; 1; 2; 0], 10^(-7), 30)
disp("===== Versão 2 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v2(A3, [1; 5; 1; 2; 0], 10^(-7), 30)
```

```
  "===== Versão 1 ====="
 lambda  =
   25.874460
 x1  =
   1.
   0.7322711
   0.9517127
   0.9614193
   0.9888749
 k  =
   13.
 n_erro  =
   6.849D-08

  "===== Versão 2 ====="
 lambda  =
   25.874461
 x1  =
   0.4797845
   0.3513323
   0.4566169
   0.4612740
   0.4744468
 k  =
   12.
 n_erro  =
   9.716D-08
```

Dessa vez, a segunda versão levou a melhor sobre a primeira, e também por 1 iteração.

Agora, vamos testar com uma matriz cujo maior autovalor em módulo é negativo. Considere a seguinte matriz:

```
A4 = [3/2 1/2 0; 1/2 3/2 0; 0 0 -3]
```

```
A4  =
  1.5   0.5   0.
  0.5   1.5   0.
  0.    0.   -3.
```

Seus autovalores são 1, 2 e **-3**. Vamos ver se nossas funções funcionarão corretamente:

```
disp("===== Versão 1 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v1(A4, [1; 2; 3], 10^(-5), 20)
disp("===== Versão 2 =====")
[lambda, x1, k, n_erro] = Metodo_potencia_v2(A4, [1; 2; 3], 10^(-5), 20)
```

```
 "===== Versão 1 ====="
lambda  =
 -3.
x1  =
  0.0001504
  0.0001504
  1.
k  =
  20.
n_erro  =
  0.0003759

 "===== Versão 2 ====="
lambda  =
 -2.9999995
x1  =
  0.0001504
  0.0001504
  1.0000000
k  =
  20.
n_erro  =
  0.0005316
```

Perfeito! As funções retornaram -3, e no mesmo número de iterações!

Para finalizar, vamos fazer um teste de desempenho com essas duas funções. Para isso, vamos gerar uma matriz grande (3000x3000) e simétrica (para garantir autovalores reais) e aplicá-la a ambas nossas funções do método da potência. Com isso, veremos o número de iterações necessário e o tempo de execução de cada uma.

```
// Gerando uma matriz aleatória simétrica 3000x3000
rand("seed", 0)
n = 3000;
A5 = 200 * rand(n, n) - 100;
```

```
A5 = (A5 + A5')/2;

// Calculando o tempo de execução de cada uma
tic()
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A5, rand(n, 1), 10^(-5), 10000);
v1_time = toc()
tic()
[lambda, x1, k2, n_erro] = Metodo_potencia_v2(A5, rand(n, 1), 10^(-5), 10000);
v2_time = toc()
```

```
 v1_time  =
   25.532164
 v2_time  =
   20.407951
```

Podemos ver que as duas funções levaram quase o mesmo tempo, indicando que elas podem ser semelhantemente eficientes.

Vamos ver quantas iterações cada uma levou.

```
k1, k2
```

```
 k1  =
   4197.
 k2  =
   4084.
```

Nesse caso, a versão 1 levou um pouco mais de iterações, mas nada muito expressivo. Portanto, é possível inferir que essas duas funções podem ser tomadas como equivalentes.

Exercício 4

Primeiramente, vamos testar isso com uma matriz aleatória:

```
// Criando uma matriz simétrica aleatória
rand("seed", 0)
A6 = round(20 * rand(5, 5) - 10);
A6 = (A6 + A6')/2
```

```
 A6  =
  -6.    4.   -4.5  -4.   -0.5
   4.    7.    3.5   1.5   0.
  -4.5   3.5   5.   -6.   -2.5
  -4.    1.5  -6.    8.   -0.5
  -0.5   0.   -2.5  -0.5  -3.
```

Estes são seus autovalores:

```
spec(A6)
```

```
ans  =
 -11.496235
 -3.1700789
  4.0558354
  8.5881460
  13.022332
```

Agora, vamos tentar encontrar seus autovalores procurando nas proximidades dos centros dos discos de Gershgorin dessa matriz, que são justamente os valores de sua diagonal.

```
for i = 1:5
    [lambda, x1, k, n_erro] = Potencia_deslocada_inversa(A6, [1; 2; 3; 4; 5],␣
 ↪10^(-5), A6(i,i), 20);
    disp(lambda)
end
```

```
  -3.1700789

   8.5881460

   4.0558354

   8.5881460

  -3.1700789
```

Podemos ver que alguns autovalores não foram localizados e outros vieram repetidos. Isso ocorreu porque os discos dessa matriz se sobrepõem, fazendo com que alguns autovalores estejam bem próximos de dois de seus centros. Com isso, esses são retornados mais de uma vez enquanto que outros que também estão dentro desses discos, mas não tão próximos dos centros, não são localizados pelo algoritmo.

Dessa forma, para termos a garantia de encontrarmos todos os autovalores da matriz com esse método, precisamos que seus discos de Gershgorin sejam todos disjuntos. Assim, abaixo temos um exemplo de uma matriz que segue essa restrição.

```
A7 = [-8 0 1 0 0;
      0 2 0 -1 0;
      1 0 12 0 -1
      0 -1 0 -3 2;
      0 0 -1 2 7]
```

```
A7  =
  -8.    0.    1.    0.    0.
   0.    2.    0.   -1.    0.
   1.    0.   12.    0.   -1.
   0.   -1.    0.   -3.    2.
   0.    0.   -1.    2.    7.
```

Vejamos seus autovalores.

```
spec(A7)
```

```
 ans  =
  -8.0500505
  -3.5608702
   2.1663483
   7.1946264
   12.249946
```

Agora, vamos ver se a busca focando nos centros desses discos resultará em todos os 5 autovalores.

```
for i = 1:5
    [lambda, x1, k, n_erro] = Potencia_deslocada_inversa(A7, [1; 2; 3; 4; 5],␣
 ↪10^(-5), A7(i,i), 20);
    disp(lambda)
end
```

```
  -8.0500505

   2.1663483

   12.249946

  -3.5608702

   7.1946264
```

Funcionou! Desse modo, concluímos que esse método é relativamente eficiente para o cálculo dos autovalores de uma matriz, mas que ele encontrará todos os autovalores sem nenhuma dúvida apenas quando todos os discos dessa matriz forem disjuntos.

Exercício 5

Experimento 1: Dois autovalores dominantes com sinais opostos

O que será que acontece com o método da potência se tivermos uma matriz com dois autovalores dominantes, mas com sinais opostos? Vamos experimentar.

Consideremos a seguinte matriz, cujos autovalores são 1, -3 e 3:

```
A8 = [2 -1 0; -1 2 0; 0 0 -3]
```

```
 A8  =
   2.   -1.    0.
  -1.    2.    0.
   0.    0.   -3.
```

Vamos ver o que nossa função vai retornar como resposta para o maior autovalor em módulo dessa matriz.

```
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A8, [0.5; -0.5; 1], 10^(-5), 20)
```

```
lambda  =
 -3.
x1  =
  0.5
 -0.5
  1.
k1  =
  20.
n_erro  =
  1.
```

Interessante... O autovalor encontrado está, de certa forma, correto. No entanto, o vetor não consegue convergir.

Vamos testar com um outro vetor inicial.

```
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A8, [-0.5; 1; -0.5], 10^(-5), 20)
```

```
lambda  =
  3.0000000
x1  =
 -1.0000000
  1.
 -0.6666667
k1  =
  20.
n_erro  =
  1.3333333
```

Agora ele foi para o outro autovalor, mesmo com o vetor ainda não convergindo.

Os autovalores encontrados nesses dois experimentos não foram aleatórios: o vetor inicial do primeiro está mais próximo do autovetor correspondente ao autovalor -3, que é $[0; 0; 1]$, enquanto que o vetor inicial do segundo está mais perto do autovetor correspondente ao autovalor 3, que é $[-1; 1; 0]$. Dessa forma, temos que, nesse caso, o algoritmo não consegue encontrar o autovetor dominante, mas ainda é capaz de descobrir um dos autovalores dominantes, e qual ele encontrará dependerá das distâncias do vetor inicial aos autovetores dominantes.

Experimento 2: Dois autovalores dominantes iguais com autovetores diferentes

E se tivermos novamente dois autovalores dominantes, mas dessa vez com mesmo sinal e com autovetores correspondentes distintos? Vamos experimentar.

Considere a matriz abaixo.

```
A9 = [3 0 0; 0 1 0; 0 0 3]
```

```
A9  =
  3.   0.   0.
  0.   1.   0.
  0.   0.   3.
```

Seus autovalores são 1, 3 e 3, mas seus autovetores mais básicos são a base canônica, com os correspondentes aos autovalores 3 sendo e_1 e e_3.

Vamos ver o que o método da potência retorna nesse caso. Façamos alguns experimentos com vetores iniciais diferentes:

```
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A9, [1; 0.5; 0], 10^(-5), 20)
```

```
lambda  =
  3.
x1  =
  1.
  0.0000028
  0.
k1  =
  11.
n_erro  =
  0.0000056
```

```
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A9, [0; 0.5; 1], 10^(-5), 20)
```

```
lambda  =
  3.
x1  =
  0.
  0.0000028
  1.
k1  =
  11.
n_erro  =
  0.0000056
```

```
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A9, rand(3, 1), 10^(-5), 20)
```

```
lambda  =
  3.
x1  =
  0.6054664
  0.0000022
  1.
k1  =
  12.
n_erro  =
  0.0000044
```

Todos os testes encontraram corretamente o autovalor, mas vamos analisar os resultados para os autovetores. No primeiro teste, ao iniciar o algoritmo com um vetor próximo de e_1, ele conseguiu convergir para o autovetor e_1. O mesmo aconteceu com o autovetor e_3 no segundo teste. No entanto, ao iniciar com um vetor aleatório, ele acaba convergindo para alguma combinação linear

desses autovetores, que, pelos teoremas da Álgebra Linear, também é autovetor para esse mesmo autovalor. Desse modo, nessa situação, parece que esse algoritmo sempre consegue encontrar os autovalores e algum autovetor, apesar de esse autovetor encontrado ser um pouco incontrolável.

Experimento 3: Autovalor próximo ao dominante

Por último, o que ocorre se o segundo maior autovalor em módulo for muito próximo do autovalor dominante? Sua proximidade afetará o cálculo do autovetor dominante? Vamos descobrir.

Considere a matriz abaixo, cujos autovalores são, 1, 2.99 e 3:

```
A10 = [2 1 0; 1 2 0; 0 0 2.99]
```

```
A10  =
  2.   1.   0.
  1.   2.   0.
  0.   0.   2.99
```

O autovetor correspondente ao autovalor 2.99 é $[0; 0; 1]$, e o correspondente ao 3 é $[1; 1; 0]$. Vamos ver o que ocorre com nossa função do método da potência:

```
rand("seed", 0)
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A10, rand(3, 1), 10^(-5), 20)
```

```
lambda  =
  2.9999936
x1  =
  0.9999979
  1.
  0.0004392
k1  =
  12.
n_erro  =
  0.0000042
```

```
rand("seed", 1)
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A10, rand(3, 1), 10^(-5), 20)
```

```
lambda  =
  2.99
x1  =
  0.4923651
  0.4923651
  1.
k1  =
  20.
n_erro  =
  0.0016412
```

```
rand("seed", 2)
[lambda, x1, k1, n_erro] = Metodo_potencia_v1(A10, rand(3, 1), 10^(-5), 20)
```

```
 lambda  =
   3.0000000
 x1  =
   1.
   1.0000000
   0.4891694
 k1  =
   20.
 n_erro  =
   0.0016360
```

Podemos ver pelos exemplos que, dependendo do vetor inicial, o algoritmo pode acertar o autovalor dominante, convergindo para 3, mas também pode acabar errando e chegando ao 2.99, o que não era desejável. Além disso, vemos que, nos dois últimos, os vetores encontrados não foram exatamente os autovetores dos autovalores encontrados, mas sim combinações lineares dos autovetores do 2.99 e do 3, o que, nesse caso, diferentemente do experimento anterior, não é correto, pois os autovalores são diferentes, apesar de serem próximos. Portanto, concluímos que esse método é sensível a autovalores numericamente próximos.