Questão 1 -

Vamos começar implementando a primeira versão do algoritmo, seguindo o pseudocódigo proposto.

```
function [lambda, x1, k, n_erro] = metodo_potencia_v1(A, x0, epsilon, M)
    k = 0;
    x0 = x0 / max(abs(x0)); // Normaliza x0 pela coordenada de maior módulo
    x1 = A * x0; // Aproximação do autovetor dominante
    n_erro = epsilon + 1; // Inicializa o erro com um valor que entre no loop

    // se a primeira iteração retornar um erro menor, não executa o loop
    while k <= M && n_erro >= epsilon
        lambda = max(abs(x1)); // Aproximação do autovalor dominante
        x1 = x1 / lambda; // Normaliza x1
        n_erro = norm(x1 - x0, 'inf'); // Calcula o erro
        x0 = x1;
        x1 = A * x0;
        k = k + 1;
    end

    if k > M
        disp('O método da potência não convergiu dentro do número máximo de iterações.');
    else
        disp('O método da potência convergiu com sucesso.');
    end

    lambda = max(abs(x1)); // Autovalor dominante
    x1 = x1 / lambda; // Autovetor unitário correspondente a lambda
endfunction
```

Então, implementei a segunda versão:

```
function [lambda, x1, k, n_erro] = metodo_potencia_v2(A, x0, epsilon, M)
    k = 0;
    x0 = x0 / norm(x0, 2); // Normaliza x0 pela norma 2
    x1 = A * x0; // Aproximação do autovetor dominante
    n_erro = epsilon + 1; // Inicializa o erro com um valor que entre no loop

    while k <= M && n_erro >= epsilon
        lambda = x1' * x0; // Quociente de Rayleigh
        if lambda < 0
            x1 = -x1; // Mantém x1 com o mesmo sentido de x0
        end
        x1 = x1 / norm(x1, 2); // Normaliza x1
        n_erro = norm(x1 - x0, 2); // Calcula o erro
        x0 = x1;
        x1 = A * x0;
        k = k + 1;
    end

    if k > M
        disp('O método da potência não convergiu dentro do número máximo de iterações.');
    else
        disp('O método da potência convergiu com sucesso.');
    end

    lambda = x1' * x0; // Autovalor dominante
    x1 = x1 / norm(x1, 2); // Autovetor unitário correspondente a lambda
endfunction
```

Questão 2 -

Aqui implementamos o método da potência deslocada com iteração inversa, conforme o pseudocódigo:

```
function [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, x0, epsilon, alfa, M)
    k = 0;
    x0 = x0 / norm(x0, 2); // Normaliza x0 pela norma 2
    n_erro = epsilon + 1; // Inicializa o erro com um valor que entre no loop

    // Decomposição LU de A - alfa * I
    [L, U, P] = lu(A - alfa * eye(size(A)));

    while k <= M && n_erro >= epsilon
        x1 = resolve_com_LU(L, U, x0, P); // Resolve o sistema com a função Resolve_com_LU
        x1 = x1 / norm(x1, 2); // Normaliza x1
        lambda = x1' * A * x1; // Quociente de Rayleigh; x1 é unitário
        if x1' * x0 < 0
            x1 = -x1; // Mantém x1 com o mesmo sentido de x0
        end
        n_erro = norm(x1 - x0, 2); // Calcula o erro
        x0 = x1;
        k = k + 1;
    end

    if k > M
        disp('O método da potência deslocada com iteração inversa não convergiu dentro do número máximo de iterações.');
    else
        disp('O método da potência deslocada com iteração inversa convergiu com sucesso.');
    end

    lambda1 = x1' * A * x1; // Autovalor de A mais próximo de alfa
endfunction
```

Lambda foi calculado usando o Rayleigh, já que temos o autovetor associado. Vale mencionar também que foi usado a decomposição LU para aumentar a velocidade do algoritmo. Para resolução do sistema, foi usada a implementação feita na atividade prática anterior, como segue:

```
////////////////////////////////////////////////////////////////////////////
//Variáveis de saída:
//X: as colunas são soluções do sistema linear Axi = bi, na mesma ordem de B
////////////////////////////////////////////////////////////////////////////
function [X]=resolve_com_LU(L, U, B, varargin)
    [n]=size(L,1);
    [n_b] = size(B,2); //conta quantos valores de b estão sendo passados na matriz B

    if nargin < 3 //detecta a matriz de permutação P foi passada
        P = eye(n,n); //caso não tenha sido, define P como a identidade
    else
        P = varargin(1); //caso tenha, define P como a matriz passada no 3 elemento
    end

    // aplica a matriz de permutação em B
    // já que PLUX = B, levando a LUX = PB
    // B antes e depois da permutação são chamados de B, mas são coisas diferentes.
    B = P * B;

    for j=1:n_b //percorre todos os vetores b em B
        y=zeros(n,1); //zera o vetor y a cada iteração
        b = B(1:n,j); //escolhe o vetor b em B, de acordo com j
        y(1)=b(1)/L(1,1); //define o primeiro elemento do vetor y
        for i=2:n
            y(i)=(b(i)-L(i,1:i-1)*y(1:i-1)); //define os outros elementos do vetor y
            // seleciona o valor em b, então subtrai o vetor correspondente aos elementos
            // da linha abaixo da diagonal multiplicado pelos valores já encontrados de y
        end
        B(1:n,j) = y; //substitui o valor de y na propria matriz B para economizar memória
    end
    Y = B; //define a nova matriz como Y, contendo todos os vetores y encontrados

    for j=1:n_b //percorre todos os vetores y em Y
        x=zeros(n,1); //zera o vetor x a cada iteração
        y = Y(1:n,j); //escolhe o vetor y em Y, de acordo com j
        x(n)=y(n)/U(n,n); //define o último elemento do vetor x
        for i=n-1:-1:1
            x(i)=(y(i)-U(i,i+1:n)*x(i+1:n))/U(i,i); //define os outros elementos do vetor x
            // seleciona o valor em y, então subtrai o vetor correspondente aos elementos
            // da linha acima da diagonal multiplicado pelos valores já encontrados de x
            // então divide pelo elemento da diagonal naquela linha
        end
        Y(1:n,j) = x; //substitui o valor de x na propria matriz Y para economizar memória
    end
    X = Y; //define a nova matriz como X, contendo todos os vetores x encontrados
    endfunction
```

Questão 3 -

Inicialmente, iremos definir um método para o teste das funções. Inicialmente, vamos definir um número alto (500) para o **máximo de iterações** e epsilon baixo (0.00001), para analisar apenas a **capacidade de convergir dos algoritmos**. Então, testamos as matrizes na função e iremos comparar o resultado com a função **spec()**, nativa do scilab.

As funções a serem testadas em cada algoritmo serão:

A_simetrica = [1, 2, 3; 2, 4, 5; 3, 5, 6], que é simétrica.

A_simetrica_baixa = [0.001, 0.002, 1e-15; 0.002, 0.004, 0.005; 1e-15, 0.005, 0.006], que é simétrica e possui os valores pequenos.

A_simetrica_grande = [1000, 2000, 3000; 2000, 1e+15, 5000; 3000, 5000, 6000], que é simétrica e possui os valores grandes.

matriz_identidade = [1 0 0; 0 1 0; 0 0 1], que é diagonal e possui todos os autovalores iguais.

matriz_diagonal = [-2 0 0; 0 -4 0; 0 0 -1], que é diagonal com autovalor negativo.

Iniciando os testes, definindo x0 = [0;0;0], descobrimos algo interessante:

```
--> A_simetrica = [1, 2, 3; 2, 4, 5; 3, 5, 6]
 A_simetrica  =

   1.   2.   3.
   2.   4.   5.
   3.   5.   6.

--> spec(A_simetrica)
 ans  =

  -0.5157295
   0.1709152
   11.344814

--> [lambda, x1, k, n_erro] = metodo_potencia_v1(A_simetrica, [0;0;0], epsilon, M)

  "O método da potência convergiu com sucesso."
 lambda  =

   Nan
 x1  =

   Nan
   Nan
   Nan
 k  =

   1.
 n_erro  =

   Nan
```

A função não funciona. Isso ocorre pois o algoritmo tenta realizar uam divisão por 0, na seguinte linha:

```
x0 = x0 / max(abs(x0)); // Normaliza x0 pela coordenada de maior módulo
```

Uma solução simples, é adicionar um número muito pequeno ao vetor x0, fazendo com que esse erro deixe de ocorrer. Essa alteração será feita posteriormente. Por agora, usaremos o vetor x0 = [1;1;1]. Refazendo o teste, agora com o novo vetor x0, temos:

```
--> x0 = [1;1;1]
 x0  =

   1.
   1.
   1.

--> [lambda, x1, k, n_erro] = metodo_potencia_v1(A_simetrica, x0, epsilon, M)

  "O método da potência convergiu com sucesso."
 lambda  =

   11.344814
 x1  =

   0.4450419
   0.8019377
   1.
 k  =

   5.
 n_erro  =

   0.0000017

--> spec(A_simetrica)
 ans  =

  -0.5157295
   0.1709152
   11.344814
```

Agora, podemos ver que o lambda retornado é exatamente o maior autovalor da matriz, contudo, o erro retornado não é 0. Isso ocorre pois a função **spec()** também tem um limite de precisão definido. Para facilitar a visualização nas próximas matrizes, irei manter apenas a saída lambda da função. Testando agora para a matriz simétrica com números pequenos, temos:

```
--> A_simetrica_baixa = [0.001, 0.002, 1e-15; 0.002, 0.004, 0.005; 1e-15, 0.005, 0.006]
 A_simetrica_baixa  =

   0.001       0.002   1.000D-15
   0.002       0.004   0.005
   1.000D-15   0.005   0.006

--> max(spec(A_simetrica_baixa))
 ans  =

   0.0102767

--> lambda = metodo_potencia_v1(A_simetrica_baixa, x0, epsilon, M)

  "O método da potência convergiu com sucesso."
 lambda  =

   0.0102768
```

Assim, vemos que a matriz convergiu com sucesso. Agora indo para a matriz simétrica com valores grandes:

```
--> A_simetrica_grande = [1000, 2000, 3000; 2000, 1e+15, 5000; 3000, 5000, 6000]
 A_simetrica_grande  =

   1000.   2000.       3000.
   2000.   1.000D+15   5000.
   3000.   5000.       6000.

--> max(spec(A_simetrica_grande))
 ans  =

   1.000D+15

--> lambda = metodo_potencia_v1(A_simetrica_grande, x0, epsilon, M)

  "O método da potência convergiu com sucesso."
 lambda  =

   1.000D+15
```

E novamente não houveram problemas. ssim, podemos concluir que números muito pequenos ou muito grandes não impactam o funcionamento dele. Agora testamos para um matriz que possui todos os autovalores iguais, nesse caso, a matriz identidade. Nesse teste, usaremos x0 = [3;3;3]:

```
--> matriz_identidade = [1 0 0; 0 1 0; 0 0 1]
 matriz_identidade  =

   1.   0.   0.
   0.   1.   0.
   0.   0.   1.

--> spec(matriz_identidade)
 ans  =

   1.
   1.
   1.

--> lambda = metodo_potencia_v1(matriz_identidade, [3;3;3], epsilon, M)

  "O método da potência convergiu com sucesso."
 lambda  =

   1.
```

Vemos que o algoritmo novamente funcionou sem problemas. Agora, testarmos para a matriz diagonal, que possui autovalor dominante negativo:

```
--> matriz_diagonal = [-2 0 0; 0 -4 0; 0 0 -1]
 matriz_diagonal  =

  -2.   0.   0.
   0.  -4.   0.
   0.   0.  -1.

--> spec(matriz_diagonal)
 ans  =

  -4.
  -2.
  -1.

--> [lambda, x1, k, n_erro] = metodo_potencia_v1(matriz_diagonal, x0, epsilon, M)

  "O método da potência não convergiu dentro do número máximo de iterações."
 lambda  =

   4.
 x1  =

   0.
   1.
   0.
 k  =

   5001.
 n_erro  =

   2.
```

Aqui detectamos um problema. Podemos ver que ela encontrou lambda = 4, que é o número simétrico ao valor certo **-4**. Contudo, isso fez com que o algoritmo não convergisse, já que o valor esperado é -4 e não 4.

Esse problema ocorreu, pois na minha implementação normalizei o vetor com o módulo, normalizando ele com o valor real, a função funciona. Então, para arrumar isso, troquei:

```
    x0 = x0 / max(abs(x0)); // Normaliza x0 pela coordenada de maior módulo
```

por:

```
    [valor, posicao] = max(abs(x0)) // Acha a posicão da coordenada de maior módulo
    x0 = x0 / x0(posicao); // Normaliza x0 pela coordenada de maior módulo
```

Em todos os lugares onde a normalização acontece. Com isso, testando novamente a função:

```
--> [lambda, x1, k, n_erro] = metodo_potencia_v1(matriz_diagonal, x0, epsilon, M)

 "O método da potência convergiu com sucesso."
lambda  =

 -4.
x1  =

 -0.0000038
  1.
 -8.731D-11
k  =

  16.
n_erro  =

  0.0000076
```

Vemos que agora ela retorna o resultado correto. Após as correções, a função ficou assim:

```
function [lambda, x1, k, n_erro] = metodo_potencia_v1(A, x0, epsilon, M)
    k = 0;
    x0 = x0 + 1e-5;
    [valor, posicao] = max(abs(x0)); // Acha a posicão da coordenada de maior módulo
    x0 = x0 / x0(posicao); // Normaliza x0 pela coordenada de maior módulo
    x1 = A * x0; // Aproximação do autovetor dominante
    n_erro = epsilon + 1; // Inicializa o erro com um valor que entre no loop

    // se a primeira iteração retornar um erro menor, não executa o loop
    while k <= M && n_erro >= epsilon
        [v, p] = max(abs(x1));
        lambda = x1(p); // Aproximação do autovalor dominante
        x1 = x1 / lambda; // Normaliza x1
        n_erro = norm(x1 - x0, 'inf'); // Calcula o erro
        x0 = x1;
        x1 = A * x0;
        k = k + 1;
    end

    if k > M
        disp('O método da potência não convergiu dentro do número máximo de iterações.');
    else
        disp('O método da potência convergiu com sucesso.');
    end

    [v, p] = max(abs(x1));
    lambda = x1(p); // Autovalor dominante
    x1 = x1 / lambda; // Autovetor unitário correspondente a lambda
endfunction
```

Agora, realizarei exatamente o mesmo teste para a segunda versão da função. Não irei mostrar aqui todos os prints, apenas para não ficar repetitivo. Irei mostrar apenas caso alguma coisa anormal aconteça.

Para x0 = [0;0;0], obtemos o mesmo erro encontrado anteriormente:

```
--> [lambda, x1, k, n_erro] = metodo_potencia_v2(A_simetrica, [0;0;0], epsilon, M)

 "O método da potência convergiu com sucesso."
lambda  =

   Nan
x1  =

   Nan
   Nan
   Nan
k  =

   1.
n_erro  =

   Nan
```

Para os autovalores negativos, a função funcionou normalmente:

```
--> [lambda, x1, k, n_erro] = metodo_potencia_v2(matriz_diagonal, x0, epsilon, M)

 "O método da potência convergiu com sucesso."
lambda  =

 -4.0000000
x1  =

  0.0000038
 -1.0000000
  8.731D-11
k  =

  16.
n_erro  =

  0.0000076
```

Então, nesse caso, precisamos apenas adicionar um valor muito pequeno ao vetor x0. Ficando com a seguinte função:

```
function [lambda, x1, k, n_erro] = metodo_potencia_v2(A, x0, epsilon, M)
    k = 0;
    x0 = x0 + 1e-5;
    x0 = x0 / norm(x0, 2); // Normaliza x0 pela norma 2
    x1 = A * x0; // Aproximação do autovetor dominante
    n_erro = epsilon + 1; // Inicializa o erro com um valor que entre no loop

    while k <= M && n_erro >= epsilon
        lambda = x1' * x0; // Quociente de Rayleigh
        if lambda < 0
            x1 = -x1; // Mantém x1 com o mesmo sentido de x0
        end
        x1 = x1 / norm(x1, 2); // Normaliza x1
        n_erro = norm(x1 - x0, 2); // Calcula o erro
        x0 = x1;
        x1 = A * x0;
        k = k + 1;
    end

    if k > M
        disp('O método da potência não convergiu dentro do número máximo de iterações.');
    else
        disp('O método da potência convergiu com sucesso.');
    end

    lambda = x1' * x0; // Autovalor dominante
    x1 = x1 / norm(x1, 2); // Autovetor unitário correspondente a lambda
endfunction
```

Agora, vamos comparar as duas funções. Para isso, usaremos parâmetros diferentes e matrizes de ordens diferentes. Faremos 2 testes:

1º teste -

A_test1 = [1 2 3 4 5; 2 6 7 8 9; 3 7 10 11 12; 4 8 11 13 14; 5 9 12 14 15];

x0 = [10;20;30;40;1];

epsilon = 0.0001;

M = 25;

```
--> A_test1 = [1 2 3 4 5; 2 6 7 8 9; 3 7 10 11 12; 4 8 11 13 14; 5 9 12 14 15];

--> x0 = [10;20;30;40;1];

--> epsilon = 0.0001;

--> M = 25;

--> [lambda, x1, k, n_erro] = metodo_potencia_v2(A_test1, x0, epsilon, M)

 "O método da potência convergiu com sucesso."
lambda  =

  44.114975
x1  =

  0.1665776
  0.3461043
  0.4660024
  0.5392449
  0.5869586
k  =

  4.
n_erro  =

  0.0000073

--> [lambda, x1, k, n_erro] = metodo_potencia_v1(A_test1, x0, epsilon, M)

 "O método da potência convergiu com sucesso."
lambda  =

  44.114984
x1  =

  0.2837978
  0.5896572
  0.7939272
  0.9187103
  1.
k  =

  4.
n_erro  =

  0.0000126
```

Podemos notar que ambas levaram o mesmo número de iterações para convergir, porém o erro da versão 2 foi menor. Além disso, uma diferença marcante é no autovetor associado. Analisando esses vetores, percebi que eles apontam para a mesma direção, mas estão em escalas diferentes. Então, essa diferença possivelmente ocorre por causa da normalização, sendo diferente em cada algoritmo. Para confirmar, apliquei a mesma normalização usada na primeira versão do algoritmo para autovetor retornado na segunda versão:

```
 x1  =

   0.1665776
   0.3461043
   0.4660024
   0.5392449
   0.5869586

--> x1/max(abs(x1))
 ans  =

   0.2837978
   0.5896572
   0.7939272
   0.9187103
   1.
```

Confirmando a minha hipótese.

2º teste -

```
A = rand(10, 10) * 10;
A_test2 = A + A';
x0 = rand(10, 1) * 1000;
epsilon = 0.0001;
M = 25;
```

```
--> [lambda, x1, k, n_erro] = metodo_potencia_v2(A_test2, x0, epsilon, M)

 "O método da potência convergiu com sucesso."
lambda  =

  104.97837
x1  =

  0.2356591
  0.3107354
  0.3661654
  0.3601216
  0.3243919
  0.2826906
  0.3426405
  0.2995116
  0.3129217
  0.3065451
k  =

  7.
n_erro  =

  0.0000899

--> [lambda, x1, k, n_erro] = metodo_potencia_v1(A_test2, x0, epsilon, M)

 "O método da potência convergiu com sucesso."
lambda  =

  104.97884
x1  =

  0.6435880
  0.8486153
  1.
  0.9834917
  0.8859179
  0.7720158
  0.9357444
  0.8179568
  0.8545930
  0.8371732
k  =

  8.
n_erro  =

  0.0000529
```

Aqui, novamente ambas convergem e os vetores são proporcionais. Contudo, agora podemos ver que a segunda versão convergiu mais rápido, necessitando apenas 7

iterações, enquanto a versão 1 precisou de 8. O erro da versão 2, nesse caso, foi maior, ou seja, não há uma relação exata entre o erro e a versão de algoritmo. Com esses testes, aparentemente a melhor escolha seria usar a versão 2, que convergiu em menos iterações, contudo, na prática, ainda precisariamos testar a velocidade de cada iteração em cada algoritmo, para poder afirmar com certeza. Então, usaremos **tic()** e **toc()** para comparação. Para isso, vamos executar:

```
tic();
[lambda, x1, k, n_erro] = metodo_potencia_v1(A_test2, x0, epsilon, M);
tempo_v1 = toc()
tic();
[lambda, x1, k, n_erro] = metodo_potencia_v2(A_test2, x0, epsilon, M);
tempo_v2 = toc()
```

O que resultou em:

```
--> tic();

--> [lambda, x1, k, n_erro] = metodo_potencia_v1(A_test2, x0, epsilon, M);

  "O método da potência convergiu com sucesso."

--> tempo_v1 = toc()
 tempo_v1  =

   0.0372050

--> tic();

--> [lambda, x1, k, n_erro] = metodo_potencia_v2(A_test2, x0, epsilon, M);

  "O método da potência convergiu com sucesso."

--> tempo_v2 = toc()
 tempo_v2  =

   0.0032070
```

Assim, podemos ver que realmente a versão 2 é mais rápida, levando 10 vezes menos tempo **(0,003s)** que a versão 1 **(0,03s)**. Para um resultado confiável, deveríamos executar isso várias vezes, contudo, como a discrepância é muito

grande, ficaremos apenas com essa. Testando para outras matrizes, o resultado é semelhante.

Questão 4 -

Aqui, começamos construindo uma matriz simétrica, escolhi a matriz [6 1 1; 1 8 1; 1 1 1], como segue:

```
A_disc  =

  6.   1.   1.
  1.   8.   1.
  1.   1.   1.
```

Então calculamos os discos de Gerschgorin. Para isso, calculamos os centros pegando os elementos da diagonal, como segue:

```
centros = diag(A_disc);
disp('Centros dos Discos de Gerschgorin:', centros);
```

```
--> disp('Centros dos Discos de Gerschgorin:', centros);

  "Centros dos Discos de Gerschgorin:"

   6.
   8.
   1.
```

E calculamos os raios a partir da soma absoluta das linhas, com exceção do elemento da diagonal:

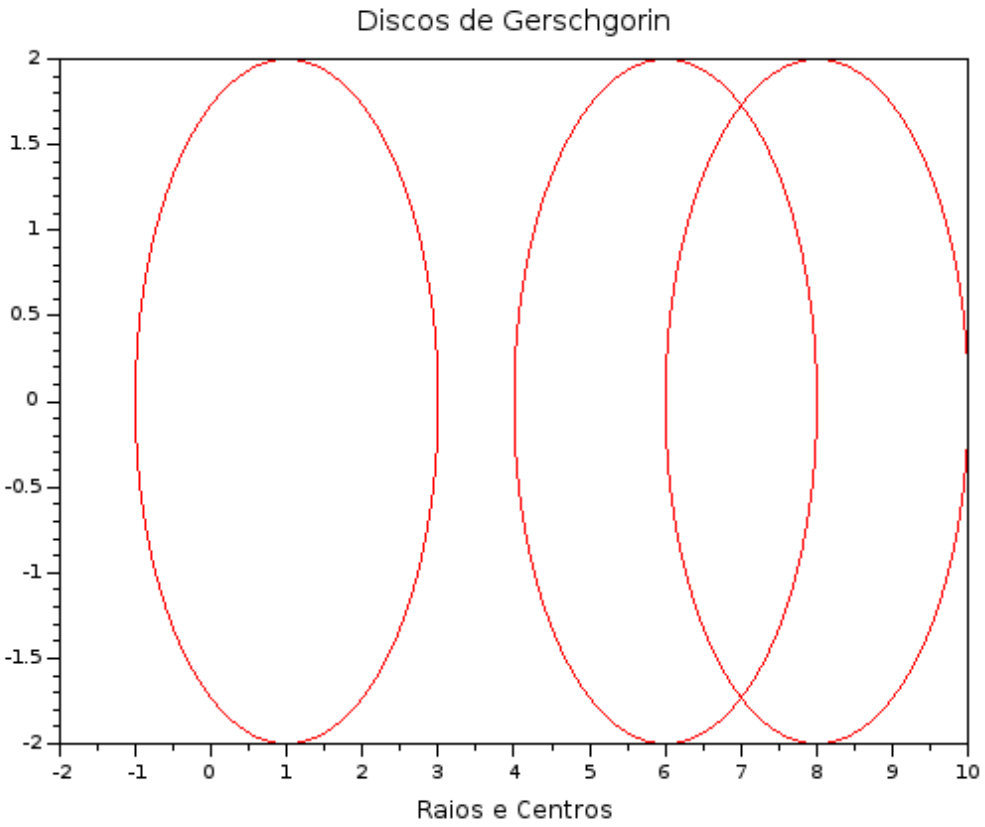
```
raios = sum(abs(A_disc ), "r")' - abs(centros);
disp( 'Raios dos Discos de Gerschgorin:', raios);
```

```
--> disp( 'Raios dos Discos de Gerschgorin:', raios);

 "Raios dos Discos de Gerschgorin:"

  2.
  2.
  2.
```

Agora, para melhor visualização, irei plotar os círculos:



Para uma melhor análise, poderíamos considerar a soma das colunas. Contudo, como a matriz é simétrica, essa análise será inútil. Então, com os discos apresentados, podemos achar:

Maior autovalor, colocando alfa = 10
Menor autovalor, colocando alfa = -1
Autovalor restante, colocando alfa = 5 (possivelmente)

Agora, podemos usar a função criada, usando os alfas previstos e ver se chegamos aos autovalores:

```
--> alfas = [-1, 5, 10];

--> for alfa = alfas
  >     lambda = potencia_deslocada_inversa(A_disc, [1;1;1], 0.0001, alfa, 40)
  > end

  "O método da potência deslocada com iteração inversa convergiu com sucesso."
 lambda  =

   0.7180183

  "O método da potência deslocada com iteração inversa convergiu com sucesso."
 lambda  =

   5.6414994

  "O método da potência deslocada com iteração inversa convergiu com sucesso."
 lambda  =

   8.6404823
```

Agora, usaremos a função **spec()** para achar os autovalores certos da matriz e comparar:

```
--> spec(A_disc)
 ans  =

   0.7180183
   5.6414994
   8.6404823
```

Como podemos ver, conseguimos chegar a exatamente os mesmos autovalores. Nesse nosso caso, tudo deu certo de primeira.

Todavia, vale ressaltar algumas coisas. Aqui usamos uma matriz 3x3, ou seja, é fácil encontrar o alfa que levará a cada um dos autovalores. Contudo, em matrizes maiores isso fica mais complicado, principalmente se houver muita sobreposição dos discos. Nesses casos, em matrizes não simétricas, realizar o cálculo dos discos tanto por colunas, quanto por linhas pode ajudar, mas mesmo assim pode não ser possível aproximar-se adequadamente.

Questão 5 -

Realizei os testes adicionais que eu achava interessante na questão 3. Assim, o relatório fica mais interessante de se ler e não há quebra de raciocínio. Porém, para não deixar essa questão vazia, vou realizar alguns testes na implementação da questão 2. Podemos testar para:

- alfa = 0
- alfa << autovalor
- alfa >> autovalor
- x0 = vetor de 0

Para esses testes, vamos definir uma matriz A qualquer.

```
--> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
 A  =

   1.    2.    3.    4.
   5.    6.    7.    8.
   9.    10.   11.   12.
   13.   14.   15.   16.
```

Então, analisamos os autovalores com **spec()**:

```
--> spec(A)
 ans  =

   36.209373 + 0.i
  -2.2093727 + 0.i
  -3.364D-15 + 0.i
  -2.280D-16 + 0.i
```

Assim, podemos começar a testar. Começaremos com alfa =0:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [1;1;1;1], 0.0001, 0, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

   Nan
x1  =

   Nan
   Nan
   Nan
   Nan
k  =

   1.
n_erro  =

   Nan
```

Aparentemente já achamos um problema. Antes de qualquer conclusão, iremos refazer o teste com uma matriz que possui autovalores distantes de 0. Testando para uma matriz diagonal, temos:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(matriz_diagonal, [1;1;1], 0.0001, 0, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

 -1.0000000
x1  =

   0.0000610
   3.725D-09
   1.0000000
k  =

   14.
n_erro  =

   0.0000610

--> spec(matriz_diagonal)
ans  =

  -4.
  -2.
  -1.
```

Assim, precisei analisar mais a fundo. Analisando, descobri que se tratava de um erro na resolução da LU. Isso porque um dos pivôs é um número muito próximo a 0, o que fazia a divisão dar errado. Então, como uma solução “gambiarra”, mas funcional para esse caso, resolvi somar 0.0000001 a todos os elementos da L e da U. Após essa mudança, ainda não conseguimos chegar a resposta exata, mas nos

aproximamos muito. Vale mencionar que não conseguimos nos aproximar mais, pois nossa tolerância ao erro é maior que o autovalor. Contudo, ainda conseguimos uma boa estimativa:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(matriz_diagonal, [1;1;1], 0.0001, 0, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

 -1.0000000
x1  =

  0.0000611
  3.705D-08
  1.0000000
k  =

  14.
n_erro  =

  0.0000610

--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [1;1;1;1], 0.0001, 0, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

 -7.665D-16
x1  =

 -0.5471753
  0.7113225
  0.2188811
 -0.3830282
k  =

  12.
n_erro  =

  0.0000456
```

Testando em um alfa diferente de 0, vemos que ainda ocorre um erro, mas dentro da tolerância, conforme o esperado:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [1;1;1;1], 0.0001, 30, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

  36.209317
x1  =

  0.1511509
  0.3492356
  0.5473203
  0.7454049
k  =

  7.
n_erro  =

  0.0000258
```

Agora, indo para um alfa >> autovalor, temos:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [1;1;1;1], 0.0001, 10000000000, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

  33.999998
x1  =

  0.5000000
  0.5000000
  0.5000000
  0.4999999
k  =

  1.
n_erro  =

  8.123D-08
```

Ou seja, deu problema também. Novamente analisando, algumas coisas. Primeiramente, por alfa >> elementos da matriz, a diagonal da matriz fica praticamente irrelevante. Segundo, os pivôs da U serão números muito grandes, levando aos problemas que tivemos na primeira aula prática. Contudo, aqui, não há como fazer a troca de linhas. Tentando resolver essa questão, vi que o fato dos números serem muito grandes pode ser resolvido, contudo, ainda assim o resultado dará errado. Isso porque ao resolver o problema dos números grandes, necessariamente vamos fazer com que os outros elementos percam a sua relevância. Então, a única forma que eu vejo de ajustar isso, é limitando o alfa.

Usando os discos de Gerschgorin, sabemos que nenhum autovalor será maior que o maior elemento da diagonal mais o seu raio, e nenhum autovalor será menor que o menor elemento da diagonal, menos seu raio (na parte real). Então, podemos usar isso para limitar. Fazemos então:

```
//Limitnado o alfa
centros = diag(A); //Centro dos discos de Gerschgorin
raios = sum(abs(A'), 2) - abs(centros); //Raios dos discos de Gerschgorin

limite_superior = centros + raios;
limite_inferior = centros - raios;

maior_limite_superior = max(limite_superior); //Maior valores real possivel
menor_limite_inferior = min(limite_inferior); //Menor valor real possivel

//Atualiza o alfa caso necessário
if alfa > maior_limite_superior
    alfa = maior_limite_superior;
elseif alfa < menor_limite_inferior
    alfa = menor_limite_inferior;
end
```

Com essa alteração, testando novamente:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [1;1;1;1], 0.0001, 10000000000, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

   36.209406
x1  =

   0.1511564
   0.3492384
   0.5473204
   0.7454024
k  =

   5.
n_erro  =

   0.0000249
```

Agora nossa implementação funciona. Contudo, essa alteração pode gerar a preocupação de não funcionar em outros casos agora. Mas tentando para várias matrizes, verifiquei que a função continua funcionando normalmente. Além disso, é possível demonstrar teoricamente que isso não afeta o funcionamento do algoritmo.

Como já fizemos a alteração tanto para o limite superior, quanto inferior, o caso onde alfa << autovalor, deve funcionar normalmente. Testando:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [1;1;1;1], 0.0001, -10e+10, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

 -2.2100978
x1  =

  0.7270724
  0.2832341
 -0.1606043
 -0.6044423
k  =

  11.
n_erro  =

  0.0000874
```

Como esperado, funcionou. Agora, basta testar o caso onde x0 = [0;0;0;0]:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [0;0;0;0], 0.0001, 35, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

  Nan
x1  =

  Nan
  Nan
  Nan
  Nan
k  =

  1.
n_erro  =

  Nan
```

Para variar, e me dar mais trabalho, não funcionou. Mas agora a solução é simples. Esse erro ocorre pois a norma de x0 será 0, então ocorrerá uma divisão por 0. Então, podemos simplesmente verificar a norma de x0 e adicionar um valor pequeno caso necessário:

```
if norm(x0, 2) == 0
    x0 = x0 + epsilon;
end
```

Agora, testando a função:

```
--> [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, [0;0;0;0], 0.0001, 35, 50)

 "O método da potência deslocada com iteração inversa convergiu com sucesso."
lambda1  =

   36.209379
x1  =

   0.1511547
   0.3492375
   0.5473203
   0.7454031
k  =

   4.
n_erro  =

   0.0000149
```

Ela está funcionando. Finalmente, podemos dar como finalizado nosso algoritmo, tendo ele ficado da seguinte forma:

```
function [lambda1, x1, k, n_erro] = potencia_deslocada_inversa(A, x0, epsilon, alfa, M)
    //Limitnado o alfa
    centros = diag(A); //Centro dos discos de Gerschgorin
    raios = sum(abs(A'), 2) - abs(centros); //Raios dos discos de Gerschgorin

    limite_superior = centros + raios;
    limite_inferior = centros - raios;

    maior_limite_superior = max(limite_superior); //Maior valores real possivel
    menor_limite_inferior = min(limite_inferior); //Menor valor real possivel

    //Atualiza o alfa caso necessário
    if alfa > maior_limite_superior
        alfa = maior_limite_superior;
    elseif alfa < menor_limite_inferior
        alfa = menor_limite_inferior;
    end

    if norm(x0, 2) == 0
        x0 = x0 + epsilon;
    end

    k = 0;
    x0 = x0 / norm(x0, 2); // Normaliza x0 pela norma 2
    n_erro = epsilon + 1; // Inicializa o erro com um valor que entre no loop
    // Decomposição LU de A - alfa * I
    [L, U, P] = lu(A - alfa * eye(A));

    while k <= M && n_erro >= epsilon
        x1 = resolve_com_LU(L, U, x0, P); // Resolve o sistema com a função Resolve_com_LU
        x1 = x1 / norm(x1, 2); // Normaliza x1
        lambda = x1' * A * x1; // Quociente de Rayleigh; x1 é unitário
        if x1' * x0 < 0
            x1 = -x1; // Mantém x1 com o mesmo sentido de x0
        end
        n_erro = norm(x1 - x0, 2); // Calcula o erro
        x0 = x1;
        k = k + 1;
    end

    if k > M
        disp('O método da potência deslocada com iteração inversa não convergiu dentro do número máximo de iterações.');
    else
        disp('O método da potência deslocada com iteração inversa convergiu com sucesso.');
    end

    lambda1 = x1' * A * x1; // Autovalor de A mais próximo de alfa
endfunction
```

Assim, podemos dar por finalizado o relatório. Acho que conseguimos cobrir muitas exceções e chegar a funções robustas e que funcionarão para muitas situações.

Gustavo Tironi