# Álgebra Linear Numérica
# Aula Prática 2

Pedro Henrique Coterli
Ciência de Dados e Inteligência Artificial

Março de 2024

## Exercício 1

Abaixo está o código da função implementando o algoritmo iterativo de Jacobi:

```
// Função do algoritmo iterativo de Jacobi
   // Variáveis de entrada:
      // A: matriz invertível A do sistema Ax = b
      // b: vetor b do sistema Ax = b
      // x0: vetor x inicial
      // E: tolerância para a diferença entre as iterações
      // M: número máximo de iterações
      // norm_type: norma utilizada
   // Variáveis de saída:
      // xk: vetor solução encontrado pelo algoritmo
      // final_delta: variação entre a penúltima e a última iteração
      // k: número de iterações realizadas
      // residue: erro em relação à solução exigida

function [xk, final_delta, k, residue] = Jacobi(A, b, x0, E, M, norm_type)
    // Dimensão de A
    n = size(A, 1)
    // Matriz triangular inferior de A
    L = tril(A, -1)
    // Gerando a matriz diagonal de A
    D = diag(diag(A))
    // Matriz triangular superior de A
    U = triu(A, 1)

    // Inicializando o número de iterações
    k = 0
```

```
    // Inicializando o vetor solução como o vetor inicial
    // e o vetor anterior a ele como um vetor grande
    // (para garantir que entrará no loop while)
    xk = x0
    x_previous = ones(n,1) * 1e30

    // Enquanto a variação entre as iterações for menor que a tolerância
    // e o número de iterações for menor que o máximo pedido...
    while norm(xk - x_previous, norm_type) > E && k < M
        // O x anterior passa a ser o x atual
        x_previous = xk
        // Calcula Dx
        Dx = -(L+U)*xk + b
        // Calcula o novo x
        xk = Dx ./ diag(D)
        // Aumenta em 1 o número de iterações
        k = k + 1
    end

    // Calculando a variação da última iteração
    final_delta = norm(xk - x_previous, norm_type)
    // Calculando o erro em relação à solução requerida
    residue = norm(b - A * xk, norm_type)
endfunction
```

# Exercício 2

## Item a)

Abaixo está o código implementando o algoritmo de Gauss-Seidel com o cálculo de inversa:

```
// Função do algoritmo iterativo de Gauss-Seidel com inversa
   // Variáveis de entrada:
      // A: matriz invertível A do sistema Ax = b
      // b: vetor b do sistema Ax = b
      // x0: vetor x inicial
      // E: tolerância para a diferença entre as iterações
      // M: número máximo de iterações
      // norm_type: norma utilizada
   // Variáveis de saída:
      // xk: vetor solução encontrado pelo algoritmo
      // final_delta: variação entre a penúltima e a última iteração
      // k: número de iterações realizadas
      // residue: erro em relação à solução exigida
```

```
function [xk, final_delta, k, residue] = Gauss_Seidel_inv(A, b, x0, E, M, norm_type)
    // Dimensão de A
    n = size(A, 1)
    // Matriz triangular inferior de A
    L = tril(A, -1)
    // Gerando a matriz diagonal de A
    D = diag(diag(A))
    // Matriz triangular superior de A
    U = triu(A, 1)

    // Inicializando o número de iterações
    k = 0

    // Inicializando o vetor solução como o vetor inicial
    // e o vetor anterior a ele como um vetor grande
    // (para garantir que entrará no loop while)
    xk = x0
    x_previous = ones(n,1) * 1e30

    // Enquanto a variação entre as iterações for menor que a tolerância
    // e o número de iterações for menor que o máximo pedido...
    while norm(xk - x_previous, norm_type) > E && k < M
        // O x anterior passa a ser o x atual
        x_previous = xk
        // Calculando o novo xk utilizando inversas
        xk = -inv(L+D)*U*xk + inv(L+D)*b
        // Aumenta em 1 o número de iterações
        k = k + 1
    end

    // Calculando a variação da última iteração
    final_delta = norm(xk - x_previous, norm_type)
    // Calculando o erro em relação à solução requerida
    residue = norm(b - A * xk, norm_type)

endfunction
```

## Item b)

Primeiramente, temos a função de resolução de sistemas triangulares inferiores:

```
// Função para resolver um sistema linear triangular inferior
   // Variáveis de entrada:
      // L: a matriz triangular inferior do sistema Lx = b
      // b: vetor b do sistema Lx = b
```

```
   // Variáveis de saída:
      // x: solução do sistema

function [x] = Solve_triangular_system(L, b)
    // Dimensão da L
    n = size(L,1)
    // Inicializando o vetor x
    x = zeros(n,1)
    // Calculando x por substituição
    x(1) = b(1)/L(1,1)
    for row = 2:n
        sub_factor = L(row,1:(row-1))*x(1:(row-1))
        x(row) = (b(row) - sub_factor)/L(row,row)
    end
endfunction
```

Agora, utilizando ela, a única mudança no código do algoritmo de Gauss-Seidel sem inversa em relação ao com inversa ocorre na parte do loop while, que fica assim:

```
    [...]
    // Enquanto a variação entre as iterações for menor que a tolerância
    // e o número de iterações for menor que o máximo pedido...
    while norm(xk - x_previous, norm_type) > E && k < M
        // O x anterior passa a ser o x atual
        x_previous = xk
        // Calculando o novo x utilizando a função de
        // resolução de sistemas triangulares
        xk = Solve_triangular_system(L+D, -U*xk+b)
        // Aumenta em 1 o número de iterações
        k = k + 1
    end
    [...]
```

# Exercício 3

Aplicando as três funções criadas anteriormente para resolver esse sistema, obtemos os seguintes resultados:

```
"Solução com Jacobi:"

-2.619D+24
-5.935D+24
 1.117D+25

"Variação final com Jacobi:"

 1.245D+25

"Número de iterações com Jacobi:"

 50.

"Resíduo com Jacobi:"

 6.322D+25
```

Resultados com Jacobi

```
"Solução com Gauss-Seidel com inversa:"

-1.188D+34
-9.829D+33
-3.072D+34

"Variação final com Gauss-Seidel com inversa:"

 3.920D+34

"Número de iterações com Gauss-Seidel com inversa:"

 50.

"Resíduo com Gauss-Seidel com inversa:"

 1.465D+35
```

Resultados com Gauss-Seidel com inversa

```
"Solução com Gauss-Seidel:"

-1.188D+34
-9.829D+33
-3.072D+34

"Variação final com Gauss-Seidel:"

 3.920D+34

"Número de iterações com Gauss-Seidel:"

 50.

"Resíduo com Gauss-Seidel:"

 1.465D+35
```

Resultados com Gauss-Seidel sem inversa

Assim, percebe-se que todas as funções geraram soluções divergentes. E isso ocorreu devido ao fato de as matrizes dos algoritmos nesse caso serem divergentes. Vamos conferir isso.

Primeiramente, analisemos a matriz do algoritmo de Jacobi. Calculando-a de acordo com a fórmula $M_j = -D^{-1}(L + U)$, obtemos a matriz abaixo e, a partir dela, utilizamos de funções do Scilab para encontrar seu raio espectral:

```
"Mj"

 0.    4.    -2.
 0.    0.    -2.
 3.   -0.5    0.

"Raio espectral:"

 3.2192026
```

Matriz de Jacobi e seu raio espectral

Agora, para o algoritmo de Gauss-Seidel, calculamos sua matriz com a

fórmula $M_{gs} = -(L + D)^{-1}U$ e seu raio espectral, exibidos abaixo:

```
"Mgs"

 0.    4.    -2.
 0.    0.    -2.
 0.    12.   -5.

"Raio espectral:"

 4.8989795
```

Matriz de Gauss-Seidel e seu raio espectral

Ambos os raios espectrais são maiores que 1. Portanto, essas matrizes não são convergentes, fazendo com que os algoritmos não convirjam para uma solução, exatamente como vimos anteriormente.

Agora, reordenando as equações de modo a obtermos uma matriz estritamente diagonal dominante, ficamos com a seguinte nova matriz de coeficientes:

```
A2  =

   6.   -1.   -2.
   1.   -4.    2.
   0.    2.    4.
```

Nova matriz de coeficientes

Por teorema, sabemos então que, com essa nova matriz sendo estritamente diagonal dominante, o algoritmo de Jacobi convergirá. Vamos conferir:

```
"Solução com Jacobi:"

 0.2498838
-0.2501428
 0.3751844

"Variação final com Jacobi:"

 0.0004342

"Número de iterações com Jacobi:"

 11.

"Resíduo com Jacobi:"

 0.0013172
```

Novos resultados com o algoritmo de Jacobi

Com apenas 11 iterações, nossa função obteve uma variação de menos de $10^{-4}$, convergindo como esperado.

Além disso, por teorema, essa nova propriedade da matriz também torna o algoritmo de Gauss-Seidel convergente, que converge inclusive mais rápido que o de Jacobi, como visto abaixo.

```
"Solução com Gauss-Seidel:"

 0.25
-0.2500509
 0.3750254

"Variação final com Gauss-Seidel:"

 0.0002843

"Número de iterações com Gauss-Seidel:"

 7.

"Resíduo com Gauss-Seidel:"

 0.0002543
```

Novos resultados com o algoritmo de Gauss-Seidel

Com apenas 7 iterações, o algoritmo de Gauss-Seidel alcança a mesma precisão que o de Jacobi.

# Exercício 4

## Item a)

Utilizando a função de Jacobi com o sistema indicado, obtemos o seguinte resultado:

```
"Solução:"

-20.827873
 2.
-22.827873

"Variação final:"

 48.219347

"Número de iterações:"

 25.

"Resíduo:"

 111.30075
```

Resultados com o algoritmo de Jacobi

Assim, vemos que o algoritmo está divergindo. E isso novamente se deve ao fato de a matriz do algoritmo não ser convergente, como mostrado abaixo.

```
"Mj"

 0.     0.5  -0.5
-1.     0.   -1.
 0.5    0.5   0.

"Raio espectral:"

 1.1180340
```

Matriz do algoritmo de Jacobi e seu raio espectral

Ela possui raio espectral maior que 1. Portanto, não é convergente.

### Item b)

Utilizando agora a função de Gauss-Seidel nas normas especificadas para resolver esse sistema, obtemos o seguinte resultado:

```
"Solução:"

 1.0000286
 1.9999676
-1.0000019

"Variação final:"

 0.0000935

"Número de iterações:"

 19.

"Resíduo:"

 0.0000877
```

Resultados com o algoritmo de Gauss-Seidel

O algoritmo converge, e isso ocorre pois a matriz do algoritmo nesse caso é convergente:

```
"Mgs"

 0.    0.5  -0.5
 0.   -0.5  -0.5
 0.    0.   -0.5

"Raio espectral:"

 0.5
```

Matriz do algoritmo de Gauss-Seidel e seu raio espectral

## Exercício 5

### Item a)

Utilizando o algoritmo de Gauss-Seidel para resolver o sistema especificado de acordo com o enunciado, obtemos os seguintes resultados:

```
"Solução:"

 0.9260770
-0.7804422
 0.6837019

"Variação final:"

 0.0947538

"Número de iterações:"

 8.

"Resíduo:"

 0.0436793
```

Resultados com o algoritmo de Gauss-Seidel

Observa-se que, como temos uma tolerância relativamente grande ($10^{-2}$), em pouquíssimas iterações (apenas 8), chegamos a um resultado dentro das condições determinadas. No entanto, ainda ficamos com um resíduo considerável no final do processo.

## Item b)

Ao aplicarmos o algoritmo ao novo sistema, obtemos o seguinte:

```
"Solução:"

 2.157D+41
 1.348D+41
-1.483D+41

"Variação final:"

 5.085D+41

"Número de iterações:"

 300.

"Resíduo:"

 5.162D+41
```

Novos resultados com o algoritmo de Gauss-Seidel

A solução agora diverge. Analisando o processo do algoritmo, vemos agora que a matriz do algoritmo derivada dessa nova matriz do sistema deixa de ser convergente (seu raio espectral passa a ser maior que 1), como mostrado abaixo:

```
"Mgs"

 0.    0.    2.
 0.    0.    1.25
 0.    0.   -1.375

"Raio espectral:"

 1.375
```

Matriz do algoritmo de Gauss-Seidel

## Exercício 6

Inicialmente, foi criada a função abaixo para realizar a geração de matrizes estritamente diagonal dominante:

```
// Função para gerar uma matriz estritamente diagonal dominante
   // Variáveis de entrada:
      // n: dimensão da matriz
   // Variáveis de saída:
      // A: uma matriz nxn estritamente diagonal dominante

function [A] = SDD_matrix(n)
    // Gera uma matriz nxn aleatória
    A = rand(n,n)
    // Para cada linha...
    for row = 1:n
        // Calcula a soma de todos os termos fora da diagonal
        row_sum = sum(abs(A(row,:))) - abs(A(row,row))
        // Adiciona essa soma ao elemento da diagonal para
        // garantir que ele será maior que essa soma
        A(row,row) = abs(A(row,row)) + row_sum
    end
endfunction
```

Em seguida, foram geradas matrizes quadradas de dimensões 10, 100, 1000 e 2000 com essa função e vetores $b$ compatíveis e utilizadas as funções de Gauss-Seidel com e sem o uso de inversa para encontrar a solução desses sistemas. Abaixo estão os tempos de execução de cada um desses processos. Cada linha é um dos sistemas e cada coluna uma das funções.

| " Sem inv | Com inv " |
|---|---|
| 0.0009256 | 0.0001977 |
| 0.0051821 | 0.0023509 |
| 0.1061124 | 0.7112501 |
| 0.3419395 | 5.2538725 |

Tempos de execução das funções de Gauss-Seidel

Nota-se que o algoritmo com a aplicação de inversa foi mais eficiente nos dois primeiros casos. No entanto, a partir do terceiro, a função da esquerda tornou-se muito mais eficaz. Uma possível explicação para isso é que, para sistemas lineares menores (10x10 e 100x100), o custo computacional adicional de calcular a inversa pode ser relativamente pequeno em comparação com o esforço necessário para resolver o sistema iterativamente. Entretanto, à medida que o tamanho do sistema linear aumenta, o custo do cálculo da inversa torna-se mais proibitivo e o método iterativo de solução de sistema triangular torna-se mais eficiente em termos de tempo de execução.