CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
MÉTODOS NUMÉRICOS COMPUTACIONAIS
Lista de exercícios 02

Thiago Henrique Gonçalves Mello - 201612060188 - thiagohgmello@gmail.com

Questão 1) A solução de sistemas lineares através da decomposição LU utilizou basicamente quatro funções desenvolvidas em algoritmo .m: 1) decomposição LU, na qual se transforma um sistema $Ax = b$ em $LUx = Pb$; 2) multiplicação de matrizes, desenvolvida pelo autor com o intuito de não utilizar o método já nativo à linguagem; 3) solução de SL (sistema linear) diagonal inferior, no qual a matriz A é diagonal inferior e o sistema a ser resolvido, segundo o método LU, é $Ly = Pb$, com L sendo uma das matrizes da transformação; e 4) solução de SL diagonal superior, no qual a matriz A é U e o vetor de excitação b é y, resultado da solução do SL diagonal inferior anterior.

Como subproduto, era desejável o cálculo do determinante e este, para matrizes diagonais, é simplesmente o produto dos elementos da diagonal principal. Como, pela propriedade de determinante, se $A = LU \Rightarrow \det(A) = \det(L)\det(U)$, o cálculo é realizado de forma simples através do produtório dos elementos da diagonal principal.

Toda a decomposição é realizada na função LUDecomp, na qual se verifica se o sistema é quadrado e então são determinadas as matrizes L, U e P da transformação.

Para validação da função, os sistemas lineares de exemplo apresentados nos slides foram testados, de forma que todos eles apresentaram como resposta o valor exato esperado.

Em relação à quantidade de linhas de código, a implementação da decomposição LU foi mais custosa quando comparada à todas as outras técnicas de decomposição. Grande parte desse excesso de código vem dos cálculos relacionados à pivotação.

Como os sistemas resolvidos foram demasiadamente pequenos e tentativas de geração de SL de ordens elevadas culminaram em sistemas impossíveis, não foi possível comparar tempo de execução entre decomposições, ficando tal procedimento à cargo apenas da análise de complexidade de código.

Questão 2) A decomposição de Cholesky foi significativamente mais simples de implementar computacionalmente, entretanto, seu uso é restrito às matrizes simétricas $(A = A^T)$ e definidas positivas $(v^T Av > 0, \forall\, v \neq 0)$. Uma vez atendidos os critérios, a decomposição é simples e apresenta resultados exatos.

Os produtos da decomposição são duas matrizes diagonais, sendo que uma é a transposta da outra e o produto delas resulta na matriz original. Outra vantagem que merece destaque é o fato de não alterar o vetor de excitação b.

Novamente, assim como ocorreu para o caso LU, a solução de um SL é feita através da decomposição e então solução de dois SL, sendo um diagonal superior e outro inferior. A comparação foi feita com auxílio dos exemplos devido, mais uma vez, à dificuldade em geração de sistemas factíveis, entretanto, nos casos passíveis de comparação, o método deu como resultados soluções exatas.

Questão 3) O refinamento possui implementação computacional simples e objetivo de melhorar alguma solução oriunda de métodos numéricos, podendo ser iterativos ou não. Seu cálculo se baseia em iterações que determinam, a cada laço, a diferença entre a aproximação e o valor esperado. Com base nesta divergência, aplica-se uma correção que se repete até que ou o erro seja menor que um limiar estabelecido ou que se atinja o limite de operações escolhido.

Como o MATLAB realiza operações com *doubles* naturalmente, a precisão já é grande o suficiente à ponto de dificultar a percepção de convergência mais lenta. Com isso, todos os

sistemas testados convergiam próximos à primeira iteração. Outro ponto que contribui para tal fenômeno é o tamanho do SL. Quanto menores são, mais fácil é de aplicar a correção.

Os testes foram feitos com auxílio dos slides, dos quais foram retirados os sistemas ensaiados. Todos convergiram de forma consideravelmente próxima para o valor exato do SL, tornando a tarefa de comparação do método difícil. Mesmo assim, foi possível perceber que a correção se mostrou bem vinda, uma vez que erros grandes foram inseridos nas soluções iniciais e mesmo assim, após o refinamento, as soluções convergiram para os valores exatos.

Questão 4) O cálculo da inversa de uma matriz nxn foi feito através da implementação do método de LeVerrier-Faddeev. Neste, o principal produto esperado é o polinômio característico, entretanto, apresenta como subprodutos o determinante e a inversa da matriz de forma consideravelmente mais simples que o método de cofator, por exemplo.

O método é demasiadamente simples de implementar computacionalmente e apresenta resultados exatos. A validação do algoritmo foi feita através do cálculo da inversa das matrizes utilizadas em todos os testes das questões anteriores.

Por ser uma operação comum, seu uso foi perpetuado em outras funções implementadas ao longo do estudo.

Questão 5) Os métodos iterativos mostram-se bem-vindos em algumas situações como nos casos em que a matriz de coeficiente é consideravelmente esparsa. Nestes casos, os métodos iterativos convergem para a margem da solução mais rapidamente.

No caso do método de Jacobi, as matrizes das iterações são estacionárias, ou seja, não dependem de qual aproximação está se tratando. Com isso, sua determinação pode ser feita uma única vez, sendo que as operações recursivas corrigem a aproximação inicial para o valor exato esperado, desde que o raio espectral da matriz de transformação M seja menor que 1. Para determinação do raio espectral, utilizou-se o método das potências, o qual define o maior autovalor de uma matriz.

A ideia do algoritmo está em decompor a matriz A em termos de uma matriz diagonal (D), uma triangular inferior com elementos da diagonal nulos (E) e outra triangular superior com os elementos da diagonal também nulos $(\mathrm{F})(\mathrm{A} = \mathrm{D} - \mathrm{E} - \mathrm{F})$. A matriz de Jacobi é definida como $\mathrm{J} = \mathrm{D}^{-1}(\mathrm{E} + \mathrm{F})$.

Assim como todos os casos anteriores, os testes foram executados baseando-se nas matrizes dos exemplos. O algoritmo é consideravelmente simples de ser implementado computacionalmente, apresentando bons resultados com convergências consideravelmente rápidas devido ao tamanho do sistema analisado.

Quando comparado aos casos de solução exata expressos nas questões anteriores, o método de Jacobi apresenta a clara desvantagem de possuir um erro intrínseco a ele. Entretanto, como elucidado anteriormente, as técnicas que utilizam convergência quando $\mathrm{n} \to \infty$ podem ser vantajosas em casos esparsos, os quais são comuns em métodos numéricos. O erro aceitável pelo programa é um parâmetro de entrada assim como o número máximo de iterações, uma forma de proteger contra *loops* infinitos nos casos de não convergência.

A aplicabilidade de métodos iterativos ficaria claro caso algumas das características acima fossem explicitadas, entretanto, mesmo não havendo casos deste tipo nos exemplos, a convergência pode ser avaliada.

Questão 6) O método de Gauss-Seidel apresenta uma alteração na disposição das matrizes utilizadas no método de Jacobi. Esta mudança, apesar de ser analiticamente simples, apresenta a grande vantagem de já implementar a correção da aproximação nas variáveis posteriores, ou seja, caso deseja-se corrigir x_n^k, as variáveis $\mathrm{x}_1^{k+1}, \mathrm{x}_2^{k+1} \ldots \mathrm{x}_{\mathrm{n}-1}^{k+1}$ utilizadas já serão as corrigidas anteriormente. O critério de convergência segue o mesmo daquele utilizado por Jacobi, com a alteração que a matriz avaliada é $\mathrm{S} = (\mathrm{D} - \mathrm{E})^{-1}\mathrm{F}$. Devido ao fato de já implementar as correções $\mathrm{k} + 1$ já pré-calculadas na iteração k, a tendência é que o método convirja mais rápido quando

comparado ao de Jacobi. Mesmo assim, devido ao fato de $S \neq J$, pode haver casos em que o problema converge para o método de Jacobi, mas não para o de Gauss-Seidel.

Os testes aplicados à rotina foram baseados nos exemplos, uma vez que determinação de SL possíveis e determinados de ordens maiores não é uma tarefa simples. Com isso, a percepção de velocidade de convergência ficou prejudicada, entretanto, foi possível validar que o algoritmo de fato converge para os casos em que o raio espectral é maior que um.

Para uma comparação mais quantitativa, seriam necessários sistemas de ordens maiores, permitindo que a convergência ocorresse com um número maior de iterações. A vantagem deste método quando comparado aos exatos é a convergência mais rápida nos casos em que a matriz de coeficientes é esparsa.

Todas as funções implementadas para solução das questões estão no Anexo.

Anexo

1- Decomposição de Cholesky

```
function [L,Lt,flag] = CholeskyDecomp(A)
%CholeskDecomp define Cholesky decomposition from A matrix (A = LL')

[At,flag] = Transp(A);

if or(ne(At,A),(flag ~= true))
    L=[];
    flag = false;
    return
end

len = length(A);

% First column calculation
L(1,1) = sqrt(A(1,1));
for i=2:len
    L(i,1) = A(i,1)/L(1,1);
end

Lsum = 0;
for j=2:len
    for k=1:(j-1)
        Lsum = Lsum + L(j,k)^2;
    end
    L(j,j) = sqrt(A(j,j) - Lsum);
    Lsum = 0;
    for i=(j + 1):len
        for k=1:(j-1)
            Lsum = Lsum + L(i,k)*L(j,k);
        end
        L(i,j) = (A(i,j) - Lsum) / L(j,j);
        Lsum = 0;
    end
end
Lt = Transp(L);

end
```

2- Solução de SL com decomposição de Cholesky

```
function [x, flag] = CholeskySolver(A,b)
%CholeskySolver Solve a linear system using Cholesky decomposition

flag = true;

[L,Lt,flag2] = CholeskyDecomp(A);
if flag2 == false
    flag = false;
```

```
    return
end

y = LSDiagInf(L,b);
x = LSDiagSup(Lt,y);

end
```

3- Determinação da diagonal dominante

```
function [flag] = DiagDom(A)
%DiagDom determine if A is diagonal dominant

A_size = size(A);
if A_size(1) ~= A_size(2)
    flag = false;
    return
end
len = A_size(1);
Lsum = 0;

for i=1:len
    diag = abs(A(i,i));
    for j=1:len
        if j ~= i
            Lsum = Lsum + abs(A(i,j));
        end
    end
    if diag <= Lsum
        flag = false;
        return
    end
    Lsum = 0;
end
flag = true;

end
```

4- Solução de SL com algoritmo de Gauss-Seidel

```
function [xk1,count,error,flag] = GaussSeidelSolver(A,b,e,kmax)
%GaussSeidelSolver solve a LS using Gauss-Seidel iterative method

flag = true;
% Square matrix evaluation
A_size = size(A);
if A_size(1) ~= A_size(2)
    xk1 = [];
    flag = false;
    return
end
```

```
len = A_size(1);

% Variables initialization
D = zeros(len);
E = zeros(len);
F = zeros(len);

xk = zeros(len,1);
xk1 = zeros(len,1);
count = 0;
error = inf;

% First xk determination
for i=1:len
    xk(i) = b(i)/A(i,i);
end

% D, E and F matrices
for i=1:len
    for j=1:len
        if i == j
            D(i,i) = A(i,i);
        elseif i < j
            E(i,j) = -A(i,j);
        elseif i > j
            F(i,j) = -A(i,j);
        end
    end
end

% S and d matrices
[q,DEinv] = PCLevFad(D-E);
S = MatrixMulti(DEinv,F);
d = MatrixMulti(DEinv,b);

% Gauss-Seidel method
while and(error >= e, count < kmax)
    xk1 = MatrixMulti(S,xk) + d;
    count = count + 1;
    error = Linfty(xk1,xk)/abs(max(xk));
    xk = xk1;
end

end
```

5- Solução de SL com algoritmo de Jacobi

```
function [x,count,error,flag] = JacobiSolver(A,b,conv,e,kmax)
%JacobiSolver solve a LS using Jacobi iterative method

% Square matrix evaluation
A_size = size(A);
if A_size(1) ~= A_size(2)
```

```
    x = [];
    flag = false;
    return
end
len = A_size(1);

% Variables initialization
xk = zeros(len,1);
J = zeros(A_size);
c = zeros(len,1);
count = 0;
error = inf;

% J, x0 and c matrices determination
for i=1:len
    for j=1:len
        if i ~= j
            J(i,j) = -A(i,j)/A(i,i);
        end
    end
    xk(i) = b(i)/A(i,i);
    c(i) = b(i)/A(i,i);
end

% Convergence garantee method
if or(conv == "n", conv == "N")
    ratio = SpecRatio(J,"PowerMet",e,kmax);
    if abs(ratio) > 1
        flag = false;
        x = [];
        return
    end
elseif or(conv == "s", conv == "S")
    flag = DiagDom(A);
    if flag == false
        x = [];
        return
    end
else
    flag = false;
    return
end

% Jacobi method
while and(error >= e, count < kmax)
    xk1 = MatrixMulti(J,xk) + c;

    count = count + 1;
    error = Linfty(xk1,xk)/abs(max(xk));
    xk = xk1;
end
x = xk1;

end
```

6- Norma infinita de um vetor

```
function [error, flag] = Linfty(v,w)
%Finfty determine the infinity norm btween two arrays

flag = true;
lenv = length(v);
lenw = length(w);

if lenv ~= lenw
    error = 0;
    flag = false;
    return
end
error = 0;

for i=1:lenv
    e = abs(v(i) - w(i));
    if e > error
        error = e;
    end
end

end
```

7- Solução de SL diagonal inferior

```
function [x,flag] = LSDiagInf(A,b)
%LSDiagSup Solve a superior diagonal linear system

A_size = size(A);
flag = true;

if A_size(1) ~= A_size(2)
    flag = false;
    return;
end
len = A_size(1);
xsum = 0;

x = zeros(len,1);

for i=1:len
    for j=1:i
        xsum = xsum - A(i,j)*x(j);
    end
    x(i) = (xsum + b(i))/A(i,i);
    xsum = 0;
end

end
```

8- Solução de SL diagonal superior

```
function [x,flag] = LSDiagSup(A,b)
%LSDiagSup Solve a superior diagonal linear system

A_size = size(A);
flag = true;

if A_size(1) ~= A_size(2)
    flag = false;
    return;
end
len = A_size(1);
xsum = 0;

x = zeros(len,1);

for i=len:-1:1
    for j=i:len
        xsum = xsum - A(i,j)*x(j);
    end
    x(i) = (xsum + b(i))/A(i,i);
    xsum = 0;
end

end
```

9- Refinamento de SL

```
function [x,count,e,flag] = LSRefinement(A,x0,b,error, maxiter)
%LSRefinement refine a LS with x0 been the first approximation

r = b - MatrixMulti(A,x0);
c = LUSolver(A,r);
x = x0 + c;
e = Linfty(x,x0);
count = 0;

while (e > error) && (count < maxiter)
    x0 = x;
    r = b - MatrixMulti(A,x0);
    c = LUSolver(A,r);
    x = x0 + c;
    e = Linfty(x,x0);
    count = count + 1;
end
if count == maxiter
    flag = false;
    return;
end

end
```

10- Decomposição LU

```
function [L,U,P,det,flag] = LUDecomp(A)
% LUDecomp realize LU decomposition determining L, U and P matrix.

% Inicializations
flag = true;
A_size = size(A);

% Check if decomposition is possible (A is a square matrix)
if A_size(1) ~= A_size(2)
    flag = false;
    return;
end

% Initializations
det = 1;
detL = 1;
detU = 1;
count = 0;
L = zeros(A_size(1));
Laux = L;
U = A;
Uaux = U;
P = zeros(A_size(1));
len = A_size(1);
Psum = zeros(len,1);

for j=1:len

    pos = 0;
    pivot = 0;

    % P' matrix determination
    for i=1:len
        if and((abs(U(i,j)) > abs(pivot)),(Psum(i) == 0))
            pivot = U(i,j);
            pos = i;
        end
    end
    if pos == 0
        flag = false;
        break;
    end
    P(pos,j) = 1;
    Psum = sum(P,2);

    % L and U matrix determination
    for i=1:len
        if Psum(i) == 0
            m = -U(i,j) / pivot;
        else
            m = 0;
```

```
        end
        L(i,j) = -m;

        for j2=j:len
            if Psum(i) == 0
                U(i,j2) = U(i,j2) + m * U(pos,j2);
            end
        end
    end
    L(pos,j) = 1;
end

% Check if the system is determined
if flag == false
    return
end

% Build P, L and U matrix after pivonting process
for j=1:len
    for i=1:len
        if P(i,j) == 1
            auxU = U(i,:);
            auxL = L(i,:);
            Uaux(j,:) = auxU;
            Laux(j,:) = auxL;
        end
    end
end
U = Uaux;
L = Laux;
P = Transp(P);

% Determinant calculation
Paux = P;
for j=1:len
    for i=1:len
        if Paux(i,j) == 1 && i ~= j
            count = count + 1;
            auxP = Paux(j,:);
            Paux(j,:) = Paux(i,:);
            Paux(i,:) = auxP;
        end
    end
end

for i=1:len
    detL = detL * L(i,i);
    detU = detU * U(i,i);
end
det = detL*detU*(-1)^count;

end
```

11- Solução de SL com decomposição LU

```
function [x,det,flag] = LUSolver(A,b)
%LUSolver Solve a linear system using LU decomposition

flag = true;

[L,U,P,det,flag2] = LUDecomp(A);
if flag2 == false
    flag = false;
    return
end

Pb = MatrixMulti(P,b);
y = LSDiagInf(L,Pb);
x = LSDiagSup(U,y);

end
```

12- Multiplicação de matrizes

```
function [C, flag] = MatrixMulti(A,B)
% MatrixMulti realize multiplication between two matrix.

% Initializaitons
flag = true;
Asize = size(A);
Bsize = size(B);
C = zeros(Asize(1),Bsize(2));
aux = 0;

% Check if operation is possible
if Asize(2) ~= Bsize(1)
    flag = false;
    return;
end

% Matrix multiplication
for i=1:Asize(1)
    for j=1:Bsize(2)
        for k=1:Asize(2)
            aux = aux + A(i,k)*B(k,j);
        end
        C(i,j) = aux;
        aux = 0;
    end
end
end
```

13- Determinação de polinômio característico pelo método de LeVerrier-Fadeev

```
function [q,Ainv,detA,flag] = PCLevFad(A)
%PCLevFad Determine characteristic polynomial, matrix inverse and
%determinant

flag = true;
A_size = size(A);
if A_size(1) ~= A_size(2)
    Ainv = [];
    q = [];
    detA = 0;
    flag = false;
    return;
end

len = A_size(1);
Bn = eye(len);
I = eye(len);
q = zeros(1,(len + 1));
q(1) = (-1)^len;

for i=1:len
    qn = 0;
    Bn_1 = Bn;
    An = MatrixMulti(A,Bn_1);
    for k=1:len
        qn = qn + An(k,k);
    end
    q(i + 1) = qn/i;
    Bn = An - (qn/i)*I;
end

if Bn ~= zeros(len)
    flag = false;
    Ainv = [];
    detA = 0;
    return;
end

Ainv = (1/q(len + 1))*Bn_1;
detA = q(len + 1)*(-1)^len;

end
```

14- Método das potências para determinação do maior autovalor

```
function [lambda,yk] = PowerMet(A,e,imax)
%PowerMet determine the higher eigenvalue of A matrix

count = 0;
A_size = size(A);
len = A_size(1);
```

```
error = inf;

yk = Transp(randi(10,1,len));
while all(yk) == false
    yk = Transp(randi(10,1,len));
end

zk = MatrixMulti(A,yk);
a = max(abs(zk));
yk = 1/a*zk;
zk = MatrixMulti(A,yk);
lambdak1 = zk./yk;

while and(error >= e, count < imax)
    lambdak = lambdak1;
    a = max(abs(zk));
    yk = 1/a*zk;
    zk = MatrixMulti(A,yk);
    lambdak1 = zk./yk;
    error = abs(max((lambdak-lambdak1)./lambdak1));
    yk = (1/a)*zk;
    count = count + 1;
end
lambda = mean(lambdak1);

end
```

15- Cálculo do raio espectral

```
function [ratio] = SpecRatio(A,met,e,imax)
%SpecRatio determine Spectral Ratio of matrix A

if met == "PowerMet"
    ratio = PowerMet(A,e,imax);
end

end
```

16- Transposição de matriz

```
function [At] = Transp(A)
%Transp determine transposition at A matrix

A_size = size(A);

At = zeros(A_size(2),A_size(1));

for i=1:A_size(1)
    At(:,i) = A(i,:);
end
```

```
for j=1:A_size(2)
    At(j,:) = A(:,j);
end

end
```