Relatório

Aula Prática 1

Nome: Felipe Marques Esteves Lamarca

Matrícula: 211708306

Para os fins deste relatório, utilizaremos o Scilab para escrever e executar os códigos. O texto está divido em seis seções, correspondentes às questões definidas pelo professor no arquivo da Aula Prática 1.

1 Testagem da função com matrizes quadradas A e vetores b

Para a testagem, faremos uso da função Gaussian_Elimination_1(A, b), fornecida na própria lista da Aula Prática 1 (os comentários deste código foram removidos, já que ele foi fornecido na própria lista):

```
function [x, C]=Gaussian_Elimination_1(A, b)
C=[A,b];
[n]=size(C,1);
for j=1:(n-1)
 for i=(j+1):n
 C(i,j)=C(i,j)/C(j,j);
 C(i,j+1:n+1)=C(i,j+1:n+1)-C(i,j)*C(j,j+1:n+1);
 end
end

x=zeros(n,1);
x(n)=C(n,n+1)/C(n,n);
for i=n-1:-1:1
 x(i)=(C(i,n+1)-C(i,i:n)*x(i:n))/C(i,i);
end
C=C(1:n,1:n);
endfunction
```

Façamos um teste para a seguinte situação, cujos resultados para o vetor $\mathbf{x}$ e para a decomposição LU são conhecidos:

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} 4 \\ 7 \\ 8 \\ 6 \end{bmatrix}, \text{LU} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1/2 & 1 & 0 & 0 \\ 0 & -2/3 & 1 & 0 \\ 0 & 0 & -3/4 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & 3/2 & -1 & 0 \\ 0 & 0 & 4/3 & -1 \\ 0 & 0 & 0 & 5/4 \end{bmatrix}$$

De fato, o programa resolve o sistema linear de forma correta — note que a definição da função pressupõe que as entradas das matrizes L e U estejam juntas na mesma matriz C retornada:

```
--> A = [2 -1 0 0; -1 2 -1 0; 0 -1 2 -1; 0 0 -1 2];

--> b = [1; 2; 3; 4];

--> [x, C] = Gaussian_Elimination_1(A, b)
 x  =

   4.0000000
   7.0000000
   8.0000000
   6.0000000
 C  =

   2.    -1.           0.           0.
  -0.5    1.5         -1.           0.
   0.    -0.6666667    1.3333333   -1.
   0.     0.          -0.75         1.25
```

Executemos um segundo teste, cujos resultados são também conhecidos, para verificar o comportamento do programa:

$$\begin{bmatrix} 1 & 3 & 0 \\ 2 & 4 & 0 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$$

$$\text{LU} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Mais uma vez, o programa acerta o *output* sem qualquer problema:

```
--> A = [1 3 0; 2 4 0; 2 0 1];

--> b = [1; 2; 2];

--> [x, C] = Gaussian_Elimination_1(A, b)
 x  =

   1.
   0.
   0.
 C  =

   1.    3.    0.
   2.   -2.    0.
   2.    3.    1.
```

Por fim, tome-se como exemplo a seguinte situação, cujos resultados são também conhecidos:

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 2 \end{bmatrix}$$

$$\mathrm{LU} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 0 & 2/3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 0 & 3/2 & 1 \\ 0 & 0 & 4/3 \end{bmatrix}, \mathrm{x} = \begin{bmatrix} 0 \\ 2 \\ 0 \end{bmatrix}$$

Mais uma vez, o programa atesta seu funcionamento:

```
--> A = [2 1 0; 1 2 1; 0 1 2];

--> b = [2; 4; 2];

--> [x, C] = Gaussian_Elimination_1(A, b)
 x  =

   0.
   2.
   0.
 C  =

   2.     1.           0.
   0.5    1.5          1.
   0.     0.6666667    1.3333333
```

2 Testagem com matriz sugerida

Agora faremos a testagem da função a partir da matriz $A_1 = \begin{bmatrix} 1 & -2 & 5 & 0 \\ 2 & -4 & 1 & 3 \\ -1 & 1 & 0 & 2 \\ 0 & 3 & 3 & 1 \end{bmatrix}$ e do vetor $b_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, ambos sugeridos pela lista da Aula Prática 1:

```
--> A1 = [1 -2 5 0; 2 -4 1 3; -1 1 0 2; 0 3 3 1];

--> b1=[1;0;0;0];

--> [x, C] = Gaussian_Elimination_1(A1, b1)
 x  =

   Nan
   Nan
   Nan
   Nan
 C  =

   1.   -2.    5.    0.
   2.    0.   -9.    3.
  -1.   -Inf  -Inf   Inf
   0.    Inf   Nan   Nan
```

Este resultado, evidentemente, não é a resposta correta, o que não é surpreendente. Afinal, se seguirmos os passos do processo de eliminação dessa matriz, verificaremos que, em determinadas situações, será necessário executarmos permutações de linha para evitar que haja entradas nulas na posição de pivô. Nossa função, como se sabe, não admite que isso ocorra: pressupõe-se, desde o início, que os elementos na posição de pivô são sempre não nulos. Na prática, isso significa que quando o programa lida com elementos nulos na posição (j, j), ele para e retorna Nan (*Not a Number*).

3 Gaussian_Elimination_2

Notamos previamente que a função Gaussian_Elimination_1 não funciona quando há elementos nulos na posição de pivô. Nosso objetivo agora é alterar esse cenário para que haja permutação de linhas quando situações dessa natureza ocorrerem. Para isso, utilizaremos o código a seguir:

```
function [x, C]=Gaussian_Elimination_2(A, b)
C=[A,b];
[n]=size(C,1);
for j=1:(n-1)
 if C(j,j) == 0      // Caso o elemento (j,j) seja nulo
 then C([j, j+1],:) = C([j+1, j],:);    // fazemos permutacao de linha
 end
 for i=(j+1):n
 C(i,j)=C(i,j)/C(j,j);
 C(i,j+1:n+1)=C(i,j+1:n+1)-C(i,j)*C(j,j+1:n+1);
 end
end

x=zeros(n,1);
x(n)=C(n,n+1)/C(n,n);
for i=n-1:-1:1
 x(i)=(C(i,n+1)-C(i,i:n)*x(i:n))/C(i,i);
end
C=C(1:n,1:n);
endfunction
```

Note que a única diferença entre o código original e a Gaussian_Elimination_2 é que adicionamos ao *for loop* um if *statement* que verifica se o elemento na posição pivô é nulo e faz permutação de linha em caso verdadeiro. Finalmente, fazendo a testagem com a matriz A_1 e o vetor b_1, recebemos como resultado:

```
--> [x, C] = Gaussian_Elimination_2(A1, b1)
 x  =

  -0.3247863
  -0.1709402
   0.1965812
  -0.0769231
 C  =

   1.   -2.    5.    0.
  -1.   -1.    5.    2.
   2.    0.   -9.    3.
   0.   -3.   -2.    13.
```

Nesse caso, como está documentado no código, fazemos um if *statement* no qual o programa verifica se o pivô é nulo e, em caso positivo, faz uma permutação de linha. Isso, como já discutimos, é justamente o que ocorre no caso da matriz e do vetor em questão — e agora nossa função admite que a situação ocorra sem que o programa quebre. O resultado, agora, está correto.

Façamos agora a testagem para a matriz $A_2 = \begin{bmatrix} 0 & 10^{-20} & 1 \\ 10^{-20} & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}$ e o vetor $b_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$:

```
--> [x, C] = Gaussian_Elimination_2(A2, b2)
 x   =

  -1.000D+20
   0.
   1.
 C   =

   1.000D-20   1.           1.
   0.          1.000D-20    1.
   1.000D+20  -1.000D+40    1.000D+40
```

Apesar de o *output* parecer razoável, ele ainda não está correto. O tratamento da linguagem Scilab a números muito pequenos (como é o caso de 10^{-20}, bastante próximo de 0) é, em geral, diminuir a acurácia das contas de tal modo que os resultados nem sempre são os esperados. Em outras palavras, ainda que não estejamos utilizando como pivô uma entrada nula, estamos, na prática, utilizando um valor muito próximo do 0. Para resolver este problema, implementaremos a solução sugerida pela lista de exercícios — qual seja, utilizar, em primeiro lugar, os pivôs de maior valor em módulo.

4 Gaussian_Elimination_3

Vimos que a função Gaussian_Elimination_2 funciona perfeitamente no caso em que os valores nas entradas pivô estão razoavelmente distantes do 0. Por outro lado, quando esses valores estão próximos de 0, o Scilab perde a acurácia e os resultados não são exatamente os esperados. Vamos implementar uma função Gaussian_Elimination_3 que, **no momento em que nota um elemento nulo na posição de pivô**, faz uma permutação de linhas para que o novo pivô seja o maior possível:

```
function [x, C]=Gaussian_Elimination_3(A, b)
C=[A,b];
[n]=size(C,1);
for j=1:(n-1)
    if C(j,j) == 0  // caso o elemento seja nulo, cria uma variavel
   maior_pivo
    then
    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],:);   // executa uma permutacao de linha
   usando a variavel criada
end

for i=(j+1):n
 C(i,j)=C(i,j)/C(j,j);
 C(i,j+1:n+1)=C(i,j+1:n+1)-C(i,j)*C(j,j+1:n+1);;
end
end

x=zeros(n,1);
x(n)=C(n,n+1)/C(n,n);
for i=n-1:-1:1
 x(i)=(C(i,n+1)-C(i,i:n)*x(i:n))/C(i,i);
end
C=C(1:n,1:n);
endfunction
```

Uma análise mais cuidadosa da função nos permite notar o seguinte: no *for loop* relativo a j, ajustamos o if *statement* (que verifica se o pivô é nulo) adicionando a definição de uma variável *maior_pivo*, que começa como 0, e um novo *for loop* que atravessa as entradas da matriz e guarda na variável **a** a linha da matriz à qual pertence o maior pivô em valor absoluto. Com essa variável, realiza a permutação de linhas. Façamos a verificação para a matriz A_2 e o vetor b_2, enunciados na seção anterior:

```
--> [x, C] = Gaussian_Elimination_3(A2, b2)
 x  =

   1.
  -1.
   1.
 C  =

   1.           2.           1.
   1.000D-20    1.           1.
   0.           1.000D-20    1.
```

O resultado é correto: note que o sistema $A_2\mathrm{x} = b_2$ é resolvido com o vetor x retornado.

Realizemos agora um segundo teste, nesse caso para a matriz $A_3 = \begin{bmatrix} 10^{-20} & 10^{-20} & 1 \\ 10^{-20} & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}$ e o vetor $b_3 = b_2$, enunciado na seção anterior:

```
--> [x, C] = Gaussian_Elimination_3(A3, b3)
```

```
x  =

   0.
  -1.
   1.
C  =

   1.000D-20   1.000D-20   1.
   1.          1.          0.
   1.000D+20   1.         -1.000D+20
```

O problema que enfrentamos nesse caso também é de natureza computacional. Nosso if *statement* verifica, para fazer troca de linha, se há algum valor maior que 0 em módulo e faz comparações a partir desse valor inicial. No entanto, a condição para que esse if *statement* seja executado é que o pivô inicial seja 0 — o que não é o caso: evidente que $10^{-20} > 0$. Mas, do ponto de vista computacional, já discutimos o fato de que o Scilab perde a acurácia em função das aproximações e o resultado se distancia do esperado (afinal, apesar de maior, 10^{-20} é muito próximo de 0) e, portanto, o sistema linear não é resolvido corretamente. Esse problema será resolvido na próxima seção.

5 Gaussian_Elimination_4

A resolução do problema se dará através da implementação da função Gaussian_Elimination_4. Trata-se de uma função igual à Gaussian_Elimination_3, a menos do fato de que removemos a obrigatoriedade de que a permutação ocorra somente nos casos em que o pivô é 0. Agora, basta que haja um elemento maior que o pivô inicial (em posição conveniente, claro), para que seja realizada uma permutação de linhas. Nesse caso, instanciamos também uma variável P, tratando-se de uma matriz identidade na qual são realizadas as mesmas permutações que ocorrem na matriz A durante a eliminação — na prática, terminamos recebendo a matriz de permutação utilizada no processo. Verifique a função a seguir:

```
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)
    // executa o mesmo processo da Gaussian_Elimination_3, mas removendo
    a condicao de que o elemento na posicao de pivo seja nulo
    maior_pivo = abs(C(j,j))
    for i=j+1 : n
        if abs(C(i, j)) > maior_pivo then maior_pivo = C(i, j);
          a=i;
```

```
        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


for i=(j+1):n
 C(i,j)=C(i,j)/C(j,j);
 C(i,j+1:n+1)=C(i,j+1:n+1)-C(i,j)*C(j,j+1:n+1);;
end
end

x=zeros(n,1);
x(n)=C(n,n+1)/C(n,n);
for i=n-1:-1:1
 x(i)=(C(i,n+1)-C(i,i:n)*x(i:n))/C(i,i);
end
C=C(1:n,1:n);
endfunction
```

Com a função criada, vamos testá-la para a matriz A_3 e o vetor b_3, enunciados na seção anterior.

```
--> [x, C, P] = Gaussian_Elimination_4(A3, b3)
 x  =

   1.
  -1.
   1.
 C  =

   1.           2.           1.
   1.000D-20    1.           1.
   1.000D-20   -1.000D-20    1.
 P  =

   0.    0.    1.
   0.    1.    0.
   1.    0.    0.
```

O resultado obtido é justamente o esperado (note, por exemplo, que $A_3\mathrm{x} = b_3$).

6 Resolve_com_LU

A nova função que desenvolveremos se chama Resolve_com_LU. Nesse caso, ela receberá como entradas as matrizes P (a matriz de permutação), C (a matriz que combina os elementos de L e de U) e uma matriz B $m \times n$. O resultado é uma matriz X cujas colunas são soluções de sistemas $Ax_i = b_i$, em que cada solução x é dada para o vetor coluna b (da matriz B) correspondente. Tome-se, para isso, a função:

```
function [X]=Resolve_com_LU(P, C, B)
U = triu(C);  // toma os elementos da diagonal e acima dela e os
    adiciona na matriz U
L = tril(C);  // toma os elementos abaixo da diagonal e os adiciona na
    matriz L
[n]=size(C,1);

for i=1:n
    L(i, i) = 1;  // torna 1 os elementos da diagonal de L
end

X = [];            // cria uma matriz X vazia

// pega as colunas de B e resolve Ax = b (com A fatorada em L e U) para
    cada uma delas. Adiciona-se o vetor x solucao parada cada um dos
    vetores b na matriz X
for i = 1 : n
    b = B(:, i);
    x = inv(U)*inv(L)*P*b;
    X = [X, x];
end
endfunction
```

A função funciona da seguinte maneira: dada a matriz C de *input* na função, fazemos com que ela seja decomposta em duas matrizes distintas — L e U, justamente aquelas que a compõem. Nesse momento, tomamos todos os elementos abaixo da diagonal e o adicionamos à matriz L, enquanto a diagonal e os elementos acima dela são adicionadas à matriz U. A seguir, tratamos a matriz L com um for loop para adicionar 1's às entradas da diagonal de L. Finalmente, definimos a matriz A ($P^T LU$) e uma matriz vazia X na qual serão inseridos os vetores x que são solução de Ax = b, justamente a operação que o código dentro do *for loop* trata de executar: para cada coluna de B, fazemos Ax = b e inserimos o vetor x na matriz X.

Podemos, agora, prosseguir com os testes, a começar para a matriz A_1, o vetor b_1 e a matriz $B_1 = \begin{bmatrix} 2 & 4 & -1 & 5 \\ 0 & 1 & 0 & 3 \\ 2 & 2 & -1 & 1 \\ 0 & 1 & 1 & 5 \end{bmatrix}$:

```
--> [x, C, P] = Gaussian_Elimination_4(A1, b1)
 x  =

  -0.3247863
  -0.1709402
   0.1965812
  -0.0769231
 C  =

   1.   -2.           5.           0.
   0.    3.           3.           1.
   2.    0.          -9.           3.
  -1.   -0.3333333   -0.6666667    4.3333333
 P  =

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

--> [X] = Resolve_com_LU(P, C, B1)
 X  =

  -2.034188    -1.9316239    1.4529915    0.8119658
  -0.6495726   -0.7008547    0.6068376    0.4273504
   0.5470085    0.9059829   -0.2478632    1.008547
   0.3076923    0.3846154   -0.0769231    0.6923077
```

Obtivemos o resultado correto.

O próximo passo é realizar o teste para a matriz A_2, o vetor b_2 e a matriz $B_2 = \begin{bmatrix} 1 & 1 & 2 \\ 1 & -1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$:

```
--> [x, C, P] = Gaussian_Elimination_4(A2, b2)
 x  =

   1.
  -1.
   1.
 C  =

   1.           2.           1.
   1.000D-20    1.           1.
   0.           1.000D-20    1.
 P  =

   0.    0.    1.
```

```
   0.    1.    0.
   1.    0.    0.

--> [X] = Resolve_com_LU(P, C, B2)
 X  =

   0.           3.    3.
  -1.000D-20   -2.   -2.
   1.           1.    2.
```

O resultado também é correto e, por isso, realizaremos o último teste utilizando a matriz A_3, o vetor $b_3 = b_2$ e a matriz $B_3 = B_2$:

```
--> [x, C, P] = Gaussian_Elimination_4(A3, b3)
 x  =

   1.
  -1.
   1.
 C  =

   1.           2.           1.
   1.000D-20    1.           1.
   1.000D-20   -1.000D-20    1.
 P  =

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

--> [X] = Resolve_com_LU(P, C, B3)
 X  =

   0.    3.    3.
   0.   -2.   -2.
   1.    1.    2.
```

Realizados todos os testes, nota-se que as funções funcionam de modo bastante razoável, o que se verifica pelos resultados coerentes.