Questão 1 -

Para iniciar, como pede no exercício 1, testei a função fornecida em algumas matrizes quadradas para entender o seu funcionamento. Para melhor entendimento, resolvi já testar com:

- uma matriz resolvível sem troca de linhas (1)
- uma matriz de posto incompleto (2)

Então escolhi as matrizes:

(1)

1.	2.	3.
6.	3.	5.
9.	2.	7.

(2)

1.	2.	3.
1.	2.	4.
3.	9.	6.

Para todas essas, o b será calculado a partir de um x = [1; 1; 1].

Para a matriz (1) a função conseguir rodar sem problemas, obtendo o seguinte resultado:

```
--> Gaussian_Elimination_1(A, b_a)
 ans  =

   1.
   1.0000000
   1.0000000
```

Como podemos ver, ele retornou 1.0000000 para os valores de x. Contudo, acertou a resposta sem problemas.

Para a matriz (2) a função rodou, retornou um resultado repleto de Nan:

```
--> Gaussian_Elimination_1(B, b_b)
 ans  =

   Nan
   Nan
   Nan
```

Isso deve ter ocorrido pelo pivô ser 0 na segunda linha da eliminação, assim, ao tentar dividir por 0, o programa parou e retornou um vetor sem valores.

Questão 2 -

Rodando a função para a matriz pedida, obtemos:

```
--> Gaussian_Elimination_1(A1, b1)
 ans  =

   Nan
   Nan
   Nan
   Nan
```

Da mesma forma que comentado anteriormente, isso deve ocorrer pois há a necessidade de troca de linha já na segunda eliminação para resolução dessa matriz.

Questão 3 -

Implementando a modificação pedida, com a adição do seguinte trecho de código:

```
//O pivô está na posição (j,j)
if C(j,j) == 0 then //verifica se o pivô é 0

  // procura valroes não nulos nas linhas abaixo e guarda
  // a posição do primeiro não nulo na variavel index
  index = find(C(j+1:n, j))(1,1);

  C([j, j+index],:) = C([j+index, j], :); //realiza a troca das linhas
end
```

a função agora pode resolver a matriz proposta, chegando ao seguinte resultado:

```
--> Gaussian_Elimination_2(A1, b1)
 ans  =

  -0.3247863
  -0.1709402
   0.1965812
  -0.0769231
```

O que acontece, é que quando o pivô é 0, identificamos isso e resolvemos (trocamos as linhas) antes do computador tentar executar a divisão por 0. Contudo, isso fornece outro problema, que é o de que a decomposição LU, agora se trata da matriz PA, sendo P a matriz de permutação. Vale mencionar que a implementação proposta procura o próximo valor que não é 0, então para uma matriz com dois valores nulos seguidos, como:

```
1.   -2.   5.   0.
2.   -4.   1.   3.
2.   -4.   7.   2.
0.    3.   3.   1.
```

A função também funcionará. Executando para essa matriz, com o b criado a partir do vetor [1;1;1;1], temos a seguinte resposta:

```
--> Gaussian_Elimination_2(T1, b_t1)
 ans  =

   1.
   1.
   1.
   1.
```

Agora, testando a matriz A2 fornecida, temos:

```
--> Gaussian_Elimination_2(A2, b2)
 ans  =

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

Testando esse resultado vemos que ele não é correto, para isso, basta multiplicar o vetor x obtido pela matriz A2, o que deveria ser exatamente o vetor b2, o que não é

```
--> x = Gaussian_Elimination_2(A2, b2)
 x  =

  -1.000D+20
   0.
   1.

--> A2*x
 ans  =

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

o caso aqui:

que é diferente do vetor b2, que é [1;0;0]. O que está acontecendo aqui é ele escolhe como pivô o valor 10^{-20}, que não é 0, mas é muito próximo e acaba fazendo o computador não ser capaz de executar contas com precisão usando ele, gerando o ruído que alterou nosso resultado.

Questão 4 -

Implementando a modificação pedida, com a adição do seguinte trecho de código:

```
//O pivô está na posição (j,j)
if C(j,j) == 0 then //verifica se o pivô é 0

  // procura o maior valor em modulo nas linhas abaixo
  // e guarda a posição dele na variavel index
  [_, index] = max(abs(C(j+1:n, j)));

  C([j, j+index],:) = C([j+index, j], :); //realiza a troca das linhas
end
```

a função agora pode resolver a matriz proposta, chegando ao seguinte resultado:

```
--> x = Gaussian_Elimination_3(A2, b2)
 x  =

   1.
  -1.
   1.
```

Agora, o que acontece é que, ao invés de usar o valor próximo de 0 para fazer as contas, a função seleciona um valor maior, nesse caso o 1, para ser o pivô, fazendo com que o computador consiga lidar melhor com as contas.
Agora, testando para a nova matriz proposta (A3), obtemos o seguinte resultado:

```
--> x = Gaussian_Elimination_3(A3,b3)
 x  =

   0.
  -1.
   1.

--> A3*x
 ans  =

   1.
   0.
  -1.
```

Como o b3 usado era originalmente [1;0;0], podemos perceber que o resultado retornado está equivocado. Então vamos analisar a matriz para entender o que pode estar acontecendo. A matriz A3 é a seguinte:

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

Bom, como vimos anteriormente, o computador não lida bem com números muito pequenos. Aqui, podemos ver que o primeiro pivô já é um número muito pequeno, e como fizemos a função apenas alterar esse número quando o pivô é 0, ele não é alterado. Então, agora precisamos fazer ela evitar o uso de pivôs pequenos.

Questão 5 -

Implementando a modificação pedida, com a adição dos seguintes trechos de código:

```
function [x, D, P]=Gaussian_Elimination_4(A, b)
[n]=size(A,1);
I = eye(n, n); //define uma matriz identidade n x n
C=[A,b,I]; //junta as matrizes A, o vetor b e a matriz identidade criada
for j=1:(n-1)
 //O pivô está na posição (j,j)

 // procura o maior valor em modulo nas linhas abaixo
 // e guarda a posição dele na variavel index
 [_, index] = max(abs(C(j:n, j)));

 index = index - 1; //corrige o valor do index, pois o vetor procurado inclui
 C([j, j+index], :) = C([j+index, j], :); //realiza a troca das linhas
```

Aqui, foi criado um matriz C contendo a matriz A, o vetor b e uma matriz identidade, nessa ordem. Essa implementação foi feita dessa forma para que quando houver a permutação da matriz A, seja feita também a permutação na matriz identidade, levando ao final a matriz P. Com isso, foi ao final foi necessário separar as matrizes, sendo usado o seguinte código para isso:

```
D = C(1:n,1:n); //isola a matriz que contêm a decomposição LU
P = C(1:n,n+2:n*2+1); //isola a matriz P
```

A matriz P também poderia ser criada isoladamente, fazendo as mesmas operações de troca de linha na matriz C e na matriz I. Possivelmente, em alguns casos essa implementação pode até ser mais eficiente, contudo, para essa atividade resolvi fazer as operações na mesma matriz.

Com essas alterações, a função agora pode resolver a matriz A3 proposta, chegando ao seguinte resultado:

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

   1.
  -1.
   1.
 P  =

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

Ao testar o valor de x encontrado, vemos que ele leva exatamente para o vetor b3. A matriz P, indica também que foi necessário fazer a permutação da linha 1 com a linha 3 para chegar a esse resultado.

Questão 6 -

Implementando a função proposta, com a criação do seguinte código:

```
function [X]=Resolve_com_LU(C, B, varargin)
[n]=size(C,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

L = tril(C, -1) + eye(C); //define a matriz L a partir de C, note que a diagonal é definida como 1.
U = triu(C); //define a matriz U

// 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
```

Aqui, usei o ***'varargin'*** para permitir que a matriz de permutação P seja um elemento opcional, que caso não seja passado será definido como a identidade. Criei uma variável ***'n_b'*** para contar quantos valores de b foram passados e executar o *for* em todos eles. Escolhi definir a matriz L com a diagonal de 1's e a matriz U, por mais que o código pudesse ser feito diretamente na matriz C. Tentei ao máximo economizar memória ao sobrescrever matrizes.

Para executar as matrizes propostas, resolvi adicionar aos B's o vetor usado na questão onde a matriz foi definida, adicionando, por exemplo, o vetor b1 como uma coluna de B1. Fiz isso para poder comparar o resultado da Eliminação Gaussiana com a função criada nessa questão. A última coluna de X sempre representa esse vetor adicionado e deve corresponder ao x retornado na Gaussian_Elimination_4.

Executando para A1 e B1, obtivemos:

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

  -0.3247863
  -0.1709402
   0.1965812
  -0.0769231
 C  =

   2.    -4.          1.          3.
   0.     3.          3.          1.
   0.5    0.          4.5        -1.5
  -0.5   -0.3333333   0.3333333   4.3333333
 P  =

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

--> Resolve_com_LU(C, B1, P)
 ans  =

  -2.034188   -1.9316239   1.4529915   0.8119658  -0.3247863
  -0.6495726  -0.7008547   0.6068376   0.4273504  -0.1709402
   0.5470085   0.9059829  -0.2478632   1.008547    0.1965812
   0.3076923   0.3846154  -0.0769231   0.6923077  -0.0769231
```

Que é exatamente o resultado esperado. Como podemos ver, a última coluna corresponde exatamente ao vetor retornado pela eliminação gaussiana direta. Além disso, todos os outros vetores foram verificados e estão corretos.
Contudo, vale mencionar aqui que foi necessária a adição de um novo parâmetro a função para isso funcionar, correspondente a matriz de permutação, já que houve trocas de linha para criação da LU. Dessa forma, conforme comentado, o sistema passou a ser PLUX = B, ou seja, precisávamos da matriz P.

Executando para A2 e B2, obtivemos:

```
--> [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.

--> Resolve_com_LU(C, B2, P)
 ans  =

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

Que novamente corresponde ao esperado. Contudo, algo curioso é que ao executar para A3 e B3, obtivemos:

```
--> [x,C,P] = Gaussian_Elimination_4(A3, b2)
 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.

--> Resolve_com_LU(C, B3, P)
 ans  =

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

E como podemos ver, os resultados para A2 e A3 foram iguais, apesar de serem matrizes diferentes. Isso ocorre porque a diferença entre as matrizes se dá trocando o elemento (1,1) que é 0 na matriz A2 para 10^{-20}, que é um número muito próximo de 0. Devido a essa proximidade com o zero, é um número que o computador tem dificuldade de guardar, o que acaba levando os resultados a serem iguais.

Com essa atividade, podemos perceber que quando lidamos com o cálculo computacional, vários detalhes devem ser levados em conta para alcançar o resultado desejado, principalmente quando lidando com números muito pequenos, os quais o computador não consegue lidar muito bem a partir de um determinado ponto.

Gustavo Tironi