**Questão 1 -**

Implementando a função que aplica o método iterativo de Jacobi:

```
////////////////////////////////////////////////////////////////////////
// Variáveis de saída:
// xk: solução do sistema Ax=b pelo método iterativo.
// norm_dif: norma da diferença entre as duas últimas iterações.
// iter: número da iteração em que o modelo atingiu a tolerância proposta.
// norm_res: norma do resíduo do sistema, ou seja, a norma de b - Axk.
////////////////////////////////////////////////////////////////////////
function [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, M, norm_type)
    //separa A em L, D e U, com A = L + D + U
    D = diag(diag(A));
    U = triu(A - D);
    L = tril(A - D);

    //calcula a inversa da matriz diagonal D
    //como D é diagonal, sua inversa é simplesmente a inversa de cada elemento na diagonal (inv(A)nn = 1/(A)nn)
    inv_D = diag(diag(1./D));
    xk = x0;
    iter = 0;
    norm_dif = E + 1;

    //define o número maximo de iterações
    while iter <= M && norm_dif > E
        xk_old = xk; //armazena o valor anterior de xk
        xk = inv_D * (b - (L + U) * xk_old); //atualiza xk de acordo com o método de Jacobi

        norm_dif = norm(xk - xk_old, norm_type); // calcula a norma da diferença
        iter = iter + 1;
    end
    // calcula a norma do resíduo do sistema
    norm_res = norm(b - A*xk, norm_type);
end
```

Aqui, vale mencionar que para evitar o uso da função inv(), que possui uma complexidade computacional elevada, usamos o fato de D ser uma matriz diagonal para encontrar sua inversa. Isso é possível, já que por D ser diagonal, sua inversa é simplesmente a inversa de cada elemento na diagonal.

Agora, testamos essa função em uma matriz com diagonal estritamente dominante, já que assim temos certeza de que o método irá convergir. Definimos a matriz A = [4, -1, 1; -1, 4, -1; 1, -1, 4], com x = [1; 2; 1] e, consequentemente, b = [3; 6; 3]:

```
A  =                    x  =          b  =

   4.   -1.    1.          1.            3.
  -1.    4.   -1.          2.            6.
   1.   -1.    4.          1.            3.
```

Então, definindo um x0, E, M e norm_type conforme a questão 3, temos o seguinte resultado na aplicação da função:

```
--> A = [4, -1, 1; -1, 4, -1; 1, -1, 4];

--> b = [3; 6; 3];

--> x0 = [0; 0; 0];

--> E = 1e-5;

--> M = 25;

--> norm_type = 'inf';

--> [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, M, norm_type)
 xk  =

   0.9999998
   1.9999995
   0.9999998
 norm_dif  =

   0.0000014
 iter  =

   11.
 norm_res  =

   0.0000014
```

Ao aplicar a função, verificamos que as normas diferença e a norma residual são iguais. Ao analisar mais a fundo, foi testado e confirmado que essas normas estão corretas. Aparentemente, as normas são iguais devido à simetria da matriz A. Isso foi teorizado, uma vez que ao testarmos com a matriz A = [4, -1, 1; -1, 4, -1; 1, -2, 4], às normas ficaram diferentes, conforme pode ser analisado abaixo:

```
--> [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, M, 1)
 xk  =

   1.0000009
   1.9999991
   1.0000012
 norm_dif  =

   0.0000080
 iter  =

   22.
 norm_res  =

   0.0000185
```

Podemos observar que em ambas as matrizes, o método convergiu em menos de 25 iterações, resultando em normas que estão abaixo da tolerância definida. Então, reduziremos o número de iterações para analisar o comportamento da função caso a tolerância não seja alcançada antes de esgotar o limite máximo de iterações. Para ilustrar esse cenário, utilizamos a matriz A = [4, -1, 1; -1, 4, -1; 1, -2, 4], que demandou 22 iterações para alcançar a tolerância, enquanto definimos o número máximo de iterações como 15. O resultado então é o seguinte:

```
--> [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, 15, 1)
 xk  =

   0.9999576
   2.0000424
   0.9999445
 norm_dif  =

   0.0003838
 iter  =

   15.
 norm_res  =

   0.0008841
```

Assim, podemos ver que as normas retornadas são maiores que a tolerância proposta, exatamente como esperado. Vemos também que a função parou

exatamente na interação máxima definida e que a solução se encontra com 4 casas decimais de precisão, o que em alguns casos pode ser suficiente. Poderíamos alterar a tolerância caso aceitássemos um erro maior no resultado.

**Questão 2 -**

**a)** Iniciaremos implementando uma função que realiza o método de Gauss-Seidel usando a função “inv” do Scilab:

```
////////////////////////////////////////////////////////////////////////////
// Variáveis de saída:
// xk: solução do sistema Ax=b pelo método iterativo.
// norm_dif: norma da diferença entre as duas últimas iterações.
// iter: número da iteração em que o modelo atingiu a tolerância proposta.
// norm_res: norma do resíduo do sistema, ou seja, a norma de b - Axk.
////////////////////////////////////////////////////////////////////////////
function [xk, norm_dif, iter, norm_res] = gauss_seidel_inv(A, b, x0, E, M, norm_type)
    //separa A em L, D e U, com A = L + D + U
    D = diag(diag(A));
    U = triu(A - D);
    L = tril(A - D);

    //calcula a inversa da matriz diagonal D + L, usando inv()
    inv_DL = inv(D + L);

    xk = x0;
    iter = 0;
    norm_dif = E + 1;

    //define o número maximo de iterações
    while iter <= M && norm_dif > E
        xk_old = xk; //armazena o valor anterior de xk
        xk = inv_DL * (b - U * xk_old); //atualiza xk de acordo com o método de Gauss-Seidel (inversa)

        norm_dif = norm(xk - xk_old, norm_type); // calcula a norma da diferença
        iter = iter + 1;
    end
    // calcula a norma do resíduo do sistema
    norm_res = norm(b - A*xk, norm_type);
end
```

Testando essa função para a matriz que usamos inicialmente na questão 1, temos:

```
--> A = [4, -1, 1; -1, 4, -1; 1, -1, 4];

--> b = [3; 6; 3];

--> x0 = [0; 0; 0];

--> E = 1e-5;

--> M = 25;

--> [xk, norm_dif, iter, norm_res] = gauss_seidel_inv(A, b, x0, E, M, norm_type)
 xk  =

   1.0000001
   1.9999999
   1.0000000
 norm_dif  =

   0.0000020
 iter  =

   8.
 norm_res  =

   0.0000006
```

Está funcionando como esperado, chegando a solução correta. Aqui, vale mencionar que comparativamente ao método de Jacobi, o qual levou 11 iterações para alcançar a tolerância proposta, com o método de Gauss-Seidel foram necessárias apenas 8 iterações.

**b)** Agora implementaremos uma função que realiza o método de Gauss-Seidel usando a resolução do sistema linear (L+D) * xk+1 = −U * xk + b para fazer as iterações. Para tal, resolvi criei uma função para resolver sistemas em que a matriz dos coeficientes é triangular inferior:

```
////////////////////////////////////////////////////////////////////////////
// Variáveis de saída:
// xk: Solução do sistema linaer
// Obs: A matrix L de entrada deve ser triangular inferior
////////////////////////////////////////////////////////////////////////////
function x = linear_solver(L, b)
        n = length(b);
        x = zeros(n, 1);

        x(1) = b(1) / L(1, 1); //calcula o primeiro elemento de x diretamente
        for i = 2:n
            x(i) = (b(i) - L(i, 1:i-1) * x(1:i-1)) / L(i, i); //define os outros elementos do vetor x
            // 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
    end
```

Aqui, usei um trecho do código utilizado na última questão da atividade 1. A única atenção a se tomar é que essa função só funciona para matrizes triangulares inferiores. Então, com essa função pronta, implementei a função que aplica o método de Gauss-Seidel propriamente dito:

```
////////////////////////////////////////////////////////////////////////////
// Variáveis de saída:
// xk: solução do sistema Ax=b pelo método iterativo.
// norm_dif: norma da diferença entre as duas últimas iterações.
// iter: número da iteração em que o modelo atingiu a tolerância proposta.
// norm_res: norma do resíduo do sistema, ou seja, a norma de b - Axk.
////////////////////////////////////////////////////////////////////////////
function [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M, norm_type)
    //separa A em L, D e U, com A = L + D + U
    D = diag(diag(A));
    U = triu(A - D);
    L = tril(A - D);

    LD = D + L;

    xk = x0;
    iter = 0;
    norm_dif = E + 1;

    while iter <= M && norm_dif > E
        xk_old = xk; //armazena o valor anterior de xk
        xk = linear_solver(LD, b -U * xk_old); //atualiza xk de acordo com o método de Gauss-Seidel
        norm_dif = norm(xk - xk_old, norm_type); // calcula a norma da diferença
        iter = iter + 1;
    end
    // calcula a norma do resíduo do sistema
    norm_res = norm(b - A*xk, norm_type);
end
```

Da mesma forma que fizemos em a), testaremos essa função para a matriz que usamos inicialmente na questão 1:

```
--> A = [4, -1, 1; -1, 4, -1; 1, -1, 4];

--> b = [3; 6; 3];

--> x0 = [0; 0; 0];

--> E = 1e-5;

--> M = 25;

--> [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M, norm_type)
 xk  =

   1.0000001
   1.9999999
   1.0000000
 norm_dif  =

   0.0000020
 iter  =

   8.
 norm_res  =

   0.0000006
```

Aqui, vale a comparação com o resultado obtido na primeira parte da questão. Podemos notar que não há diferença alguma entre os resultados, nem no número de iterações, nem nas normas. Assim, fica evidente que ambas as abordagens fazem o trabalho da mesma forma.

Portanto, a diferença está na velocidade com que cada função realiza o trabalho, já que, como comentado anteriormente, a função inv() do Scilab possui uma complexidade computacional elevada e em matrizes maiores ela terá dificuldades.

**Questão 3 -**

Iremos inicialmente analisar as duas funções para a matriz proposta. Não há a necessidade de testar para as três, visto que as duas implementações na questão 2 levam para os mesmos resultados. As duas funçẽos usarão os seguintes parâmetros:

```
--> A = [1 -4 2; 0 2 4; 6 -1 -2];

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

--> x0 = [0; 0; 0];

--> E = 1e-5;

--> M = 25;
```

Então, para o método de Jacobi temos:

```
--> [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, M, norm_type)
 xk  =

  -3.012D+12
  -1.552D+12
  -9.561D+10
 norm_dif  =

   2.919D+12
 iter  =

   25.
 norm_res  =

   1.633D+13
```

Para o método de Gauss-Seidel temos:

```
--> [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M
 xk  =

   5.066D+16
   7.964D+15
   1.480D+17
 norm_dif  =

   1.520D+17
 iter  =

   25.
 norm_res  =

   6.079D+17
```

Analisando, podemos ver que ambas divergiram, já que os valores ficaram muito altos. Além disso, podemos notar que o método de Gauss-Seidel resultou em valores maiores para o vetor x. Isso ocorreu, pois da mesma forma que ele converge mais rapidamente, quando a matriz é divergente, ele diverge mais rapidamente que o método de Jacobi. Para confirmar que essa matriz é divergente, podemos aumentar o número de iterações e ver se o vetor x se aproxima da solução. Testando para 100 iterações:

```
--> [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, 
 xk  =

   3.532D+67
  -2.646D+68
   2.382D+68
 norm_dif  =

   3.395D+68
 iter  =

   100.
 norm_res  =

   1.570D+69
```

Vemos que os números ficaram ainda mais distantes da solução, confirmando que a solução não está convergindo.

Então, iremos reorganizar a matriz para que ela fique com a diagonal estritamente dominante, garantindo a convergência. Fazemos isso trocando as linhas 1 - 3 e 2 -3 na nova matriz formada, resultando na seguinte matriz, com as trocas também aplicadas ao b:

```
A  =                          b  =

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

Aplicando novamente as funções nessa nova matriz, temos para o método de Jacobi:

```
--> [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, M, norm_type)
 xk  =

   0.4166671
  -0.2166662
   0.3583318
 norm_dif  =

   0.0000026
 iter  =

   18.
 norm_res  =

   0.0000053
```

Para o método de Gauss-Seidel temos:

```
--> [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M
 xk  =

   0.4166667
  -0.2166659
   0.3583330
 norm_dif  =

   0.0000038
 iter  =

   10.
 norm_res  =

   0.0000038
```

Portanto, vemos que agora ambas convergiram. Levando 18 iteraçêos para isso no Jacobi e 10 no Gauss-Seidel. Esses resultados são interessantes, já que se trata do mesmo sistema linear, contudo em um caso ele diverge e no outro ele converge.
A explicação para isso é que a convergência irá acontecer se e somente se o raio espectral da matriz do método foi menor que 1. Quando trocamos as linhas, estamos alterando as matrizes L, D e U, consequentemente, alterando a matriz do método. Portanto, como haverão elementos diferentes nas matrizes em cada caso, os autovalores serão diferentes e em um caso menores que 1 (em módulo), enquanto em outro caso ao menos um é maior (em módulo).

Usando o scilab para confirmar isso, usaremos a função spec() para analisar a matriz no método de Jacobi. Analisando a matriz sem trocas de linhas, temos os seguintes autovalores:

```
--> A = [1 -4 2; 0 2 4; 6 -1 -2];

--> D = diag(diag(A));

--> U = triu(A - D);

--> L = tril(A - D);

--> inv_D = diag(diag(1./D))
 inv_D  =

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

--> J = inv_D * (L+U)
 J  =

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

--> spec(J)
 ans  =

  -1.1579362 + 3.0037391i
  -1.1579362 - 3.0037391i
   2.3158724 + 0.i
```

Como podemos notar, realmente há valores maiores que 1, em módulo. Agora realizando a mesma análise após a troca de linhas, temos:

```
--> A = [6 -1 -2; 1 -4 2; 0 2 4];

--> D = diag(diag(A));

--> U = triu(A - D);

--> L = tril(A - D);

--> inv_D = diag(diag(1./D))
 inv_D  =

   0.1666667   0.      0.
   0.         -0.25    0.
   0.          0.      0.25

--> J = inv_D * (L+U)
 J  =

   0.     -0.1666667  -0.3333333
  -0.25    0.         -0.5
   0.      0.5         0.

--> spec(J)
 ans  =

   0.1744964 + 0.i
  -0.0872482 + 0.4808015i
  -0.0872482 - 0.4808015i
```

Agora, sem nenhum autovalor maior que 1 em módulo. Assim, mostramos o porquê a troca de linhas afetou o resultado, apesar de continuar sendo o mesmo sistema linear.

**Questão 4 -**

**a)** Testando o método de Jacobi para a matriz indicada, temos:

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

   2.  -1.   1.
   2.   2.   2.
  -1.  -1.   2.

--> b = [-1; 4; -5]
 b  =

  -1.
   4.
  -5.

--> [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, M, norm_type)
 xk  =

  -20.827873
   2.
  -22.827873
 norm_dif  =

   34.924597
 iter  =

   25.
 norm_res  =

   87.311491
```

O que está muito longe da solução real, que é x = [1; 2; -1]. Podemos analisar o comportamento do método aumentando o número de iterações:

```
--> [xk, norm_dif, iter, norm_res] = jacobi(A, b, x0, E, 100, norm_type)
 xk  =

  -42037.954
  -168153.82
   42037.954
 norm_dif  =

   168155.82
 iter  =

   100.
 norm_res  =

   336311.63
```

Aqui, vemos que a solução se aproxima do vetor x = [-1;-4;1], que ainda não é o vetor procurado. analisando a matriz do método de jacobi vemos:

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

--> D = diag(diag(A));

--> U = triu(A - D);

--> L = tril(A - D);

--> inv_D = diag(diag(1./D));

--> J = inv_D * (L+U);

--> spec(J)
 ans  =

   -5.551D-17 + 1.118034i
   -5.551D-17 - 1.118034i
   -2.601D-17 + 0.i
```

E como -5.551D-17 + 1.118034i  -5.551D-17 - 1.118034i são maiores que 1 em módulo, confirmamos que a matriz está divergindo. Isso implica que para qualquer vetor x0, o método irá divergir.

**b)** Testando o método de Gauss-Siedel para a matriz indicada, temos:

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

--> b = [-1; 4; -5];

--> x0 = [0; 0; 0];

--> E = 1e-5;

--> M = 25;

--> [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M, norm_type)
 xk  =

   0.9999988
   2.0000013
  -0.9999999
 norm_dif  =

   0.0000038
 iter  =

   24.
 norm_res  =

   0.0000036
```

Agora vemos que o método convergiu para a solução correta. Além disso, podemos ver que ele demandou 24 iterações para chegar a tolerância indicada, quase no limite que definimos. Podemos agora analisar a raio espectral da matriz deste método, já sabendo que ele converge:

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

--> D = diag(diag(A));

--> U = triu(A - D);

--> L = tril(A - D);

--> G = -inv(D + L) * U;

--> spec(G)
 ans  =

   0.  + 0.i
  -0.5 + 0.i
  -0.5 + 0.i
```

Todos os valores são menores do que 1, em módulo, como já era esperado.

**Questão 5 -**

**a)** Testando o método de Gauss-Siedel para a matriz e parâmetros indicados, temos:

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

--> b = [0.2; -1.425;2];

--> x0 = [0; 0; 0];

--> E = 1e-2;

--> M = 300;

--> [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M, norm_type)
 xk  =

   0.9015543
  -0.7988343
   0.6990286
 norm_dif  =

   0.0040412
 iter  =

   14.
 norm_res  =

   0.0025258
```

Bom, como sabemos que a solução é x = [0.9; -0.8; 0.7], podemos concluir que o método convergiu. Foram necessárias para isso apenas 18 das 300 iterações dadas como máxima.

**b)** Alterando a matriz como indicado, ou seja, subtraindo 1 no elemento $A_{13}$, e aplicando a função, ficamos com:

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

--> [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M, norm_type)
 xk  =

   2.157D+41
   1.348D+41
  -1.483D+41
 norm_dif  =

   3.726D+41
 iter  =

   300.
 norm_res  =

   5.123D+41
```

Fica evidente agora que algo mudou, apesar de modificarmos muito pouco a matriz original, foi o suficiente para alterar muito significativamente o resultado. Verificando o raio espectral do método, confirmamos que a matriz se tornou divergente:

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

--> D = diag(diag(A));

--> U = triu(A - D);

--> L = tril(A - D);

--> G = -inv(D + L) * U;

--> spec(G)
 ans  =

   0.     + 0.i
   0.     + 0.i
  -1.375  + 0.i
```

Já que com esses autovalores, o raio espectral será 1,375, em módulo. Isso nos mostra que mesmo pequenas modificações, alteram muito quando estamos trabalhando com métodos iterativos.

**Questão 6 -**

Para começar, criei uma função que gera uma matriz de tamanho definido com diagonal estritamente dominante. Para isso, inicializo a matriz com valores aleatórios entre 0 e 10 e então adiciono à diagonal a soma absoluta da sua respectiva linha:

```
////////////////////////////////////////////////////////////////////////
// Variáveis de saída:
// b: Vetor b para x sendo um vetor unitário, com os três primeiros elementos igual a 2 [2 ; 2; 2; 1; ... ; 1]
// A: Matriz quadrada nxn com valores aleatórios entre 0 e 10, com a diagonal estritamente dominante.
// Obs: a garantia da dominancia se da pela soma absoluta de todos
// os elementos da linha na diagonal daquela linha
////////////////////////////////////////////////////////////////////////
function [A, b] = matriz_diag_dominante(n)
    // gera uma matriz A com valores aleatórios entre 0 e 10
    A = rand(n, n) * 10;

    // calcula a soma dos valores absolutos de cada linha
    soma_linhas = sum(abs(A), 2);

    // adiciona uma diagonal estritamente dominante à matriz A
    A = A + diag(soma_linhas);

    // gerar vetor x e, a partir dele, o vetor b
    x = ones(n, 1);
    x(1:3) = 2;
    b = A*x
endfunction
```

Um exemplo de utilização pode ser analisado abaixo, para uma matriz 7x7:

```
--> matriz_diag_dominante(8)
 ans  =

   39.176307   4.3832764   9.9291496   3.2900535   0.7392961   7.0597066   3.7858232   1.1960808
   6.5407369   67.122218   9.7574285   4.8089468   9.4336947   7.0181696   4.6195234   7.6139997
   5.8781064   3.7921421   36.845878   3.303696    1.2863307   4.0879997   6.2873698   4.7909885
   6.0208319   7.6687161   3.0322382   37.681524   2.0190808   0.6362214   2.8785153   2.8169693
   0.453502    6.0066213   9.5195201   2.1171908   28.364981   0.6573934   3.2920487   2.3800978
   2.0294443   7.8567356   7.1278581   4.4860231   8.9286902   49.104198   4.719233    3.2942055
   7.8442738   7.3871156   1.1923701   5.9145097   4.617919    0.3315819   36.302037   2.306728
   2.6375362   5.5442603   5.0091632   6.8067427   6.2512917   3.1578356   5.5530697   39.232493
```

Com isso, criei uma função que executa ambas as funções do método de Gauss-Seidel e mede o tempo necessário para elas ocorrerem:

```
exec('gauss_seidel.sci');
exec('matriz_diag_dom.sci');

function [tempo_inv, tempo_sislin, diferenca] = tempo_execucao(tamanho_matriz)
    // gera matriz A com diagonal estritamente dominante e vetor b compatível
    [A, b] = matriz_diag_dominante(tamanho_matriz);

    // vetor inicial x0 de zeros
    x0 = zeros(tamanho_matriz, 1);

    // parâmetros para o método de Gauss-Seidel
    E = 1e-2;
    M = 500;
    norm_type = 2;

    // mede o tempo de execução do método de Gauss-Seidel com inv()
    tic();
    [xk, norm_dif, iter, norm_res] = gauss_seidel_inv(A, b, x0, E, M, norm_type);
    tempo_inv = toc();

    // mede o tempo de execução do método de Gauss-Seidel com a resolução do sistema linear
    tic();
    [xk, norm_dif, iter, norm_res] = gauss_seidel_sislin(A, b, x0, E, M, norm_type);
    tempo_sislin = toc();

    diferenca = abs(tempo_inv - tempo_sislin)


    disp('Para n igual a ' + string(tamanho_matriz));
    disp('Tempo do método de Gauss-Seidel com inv(): ' + string(tempo_inv) + ' segundos')
    disp('Tempo do método de Gauss-Seidel com sistema linear: ' + string(tempo_sislin) + ' segundos')
    disp('Diferença de tempo: ' + string(diferenca) + ' segundos')
endfunction
```

Aqui, vale mencionar que defini o número máximo de iterações para 500, a tolerância para $10^{-2}$ e norma euclidiana. A matriz A e o vetor b são gerados pela própria função, com base no tamanho definido. Aplicando essa função para os tamanhos propostos temos:

```
--> [tempo_inv, tempo_sislin, diferenca] = tempo_execucao(10);

  "Para n igual a 10"

  "Tempo do método de Gauss-Seidel com inv(): 0.00034 segundos"

  "Tempo do método de Gauss-Seidel com sistema linear: 0.001018 segundos"

  "Diferença de tempo: 0.000678 segundos"

--> [tempo_inv, tempo_sislin, diferenca] = tempo_execucao(100);

  "Para n igual a 100"

  "Tempo do método de Gauss-Seidel com inv(): 0.006416 segundos"

  "Tempo do método de Gauss-Seidel com sistema linear: 0.005136 segundos"

  "Diferença de tempo: 0.00128 segundos"
```

```
--> [tempo_inv, tempo_sislin, diferenca] = tempo_execucao(1000);
  "Para n igual a 1000"
  "Tempo do método de Gauss-Seidel com inv(): 0.084404 segundos"
  "Tempo do método de Gauss-Seidel com sistema linear: 0.155002 segundos"
  "Diferença de tempo: 0.070598 segundos"
--> [tempo_inv, tempo_sislin, diferenca] = tempo_execucao(2000);
  "Para n igual a 2000"
  "Tempo do método de Gauss-Seidel com inv(): 0.573121 segundos"
  "Tempo do método de Gauss-Seidel com sistema linear: 0.589101 segundos"
  "Diferença de tempo: 0.01598 segundos"
--> [tempo_inv, tempo_sislin, diferenca] = tempo_execucao(3000);
  "Para n igual a 3000"
  "Tempo do método de Gauss-Seidel com inv(): 2.101431 segundos"
  "Tempo do método de Gauss-Seidel com sistema linear: 1.194955 segundos"
  "Diferença de tempo: 0.906476 segundos"
--> [tempo_inv, tempo_sislin, diferenca] = tempo_execucao(5000);
  "Para n igual a 5000"
  "Tempo do método de Gauss-Seidel com inv(): 9.525215 segundos"
  "Tempo do método de Gauss-Seidel com sistema linear: 2.607061 segundos"
  "Diferença de tempo: 6.918154 segundos"
```

Analisando os resultados, vemos que até n = 2000 é preferível usar o método inv() do scilab, que está conseguindo realizar a função de maneira mais rápida. Contudo, a partir disso, o método da inversa começa a ter um tempo de execução exponencialmente maior. Ao chegarmos em um n = 5000, podemos ver que com a inversa demoramos ~10s, quanto com o sistema linear demoramos ~2,5s, ou seja, a inversa nesse ponto está demorando 4 vezes mais. Isso ocorre pois a complexidade computacional do cálculo da inversa é dado por $O(n^3)$. Aumentar ainda mais a matriz levará a um tempo de execução muito longo, então irei encerrar por aqui.