Relatório
Aula Prática 2

Nome: Felipe Marques Esteves Lamarca
Matrícula: 211708306

1 Introdução

Este relatório implementa e testa métodos iterativos de resolução de sistemas lineares, mais especificamente os métodos de Jacobi e Gauss-Seidel. De modo geral, esses são métodos através dos quais partimos de um vetor inicial qualquer, geralmente o vetor nulo, a partir do qual geramos uma série de outros vetores que compõem uma sequência. O limite dessa sequência é justamente o vetor que corresponde à solução (ou melhor, uma aproximação dela) do sistema linear em questão.

Ficaremos atentos a questões específicas de cada método e, mais especificamente, à questão do tempo de execução de cada um, quando essa análise parecer conveniente. Veremos, por exemplo, o custo computacional de se utilizar a função **inv()** do Matlab/Scilab e os limites e vantagens de se utilizar cada um dos métodos em determinadas situações.

2 Implementação das Funções

2.1 Método de Jacobi

A primeira função implementada executa o método de Jacobi para resolução de sistemas lineares. Nesse caso, encontramos o vetor x que resolve o sistema linear a partir da seguinte relação:

$$\mathrm{x}^{(k+1)} = -D^{-1}(L+U)\mathrm{x}^{(k)} + D^{-1}\mathrm{b}$$

Já sabemos que o método em questão é iterativo e, portanto, se repete diversas vezes — para encontrar o vetor $x^{(k+1)}$, partimos do vetor anterior a ele, $x^{(k)}$. A matriz D da equação é diagonal, enquanto L e U são matrizes triangular inferior e triangular superior, respectivamente.

A implementação da função se deu como segue:

```
function [xk, dif_norm, k, residuo] = Jacobi(A, b, x0, E, M, norma)

    U = triu(A, 1);
    L = tril(A, -1);
    D_inverse = diag(1./diag(A));

    xk = x0;
    k = 0;
    dif = E+1;

    while norm((dif), norma)>E & k<M
        xj = xk;
        xk = -(D_inverse)*(L+U)*xj + (D_inverse)*b;
        dif = xk-xj;
        k = k+1;
    end

    dif_norm = norm((dif), norma);
    residuo = norm((b-A*xk), norma);

endfunction
```

O significado de cada entrada e cada saída foi explicitado na própria lista da Aula Prática 2. Cabe comentar o código:

1. Criamos 3 matrizes: U, L e D_inverse, todas a partir da matriz A no *input*. A primeira é triangular superior com zeros na diagonal; a segunda é triangular inferior com zeros na diagonal; e a terceira é diagonal;

2. Definimos que o vetor xk terá, de início, o valor do vetor x0 do *input*. Além disso, k, que corresponde ao número de iterações, começa em 0 — claro: não há iterações antes do *loop*. A variável dif, redefinida no *loop*, simplesmente é definida inicialmente como E+1 para que consigamos entrar no *loop*;

3. O *loop* em si simplesmente realiza o processo de resolução de sistema linear de acordo com o método de Jacobi. No processo, a diferença entre os vetores xk e o predecessor imediato a ele é calculada a cada iteração, ao mesmo tempo que a variável k é acrescida em uma unidade a cada vez que uma ação do *loop* é executada;

4. Finalmente, criamos as variáveis dif_norm e residuo para que elas guardem, respectivamente, a norma da diferença entre $||x_k - x_{k-1}||$ e o resíduo $||r_k|| = ||b - Ax_k||$.

2.2 Método de Gauss-Seidel (usando **inv()**)

O método de Gauss-Seidel é um pouco diferente do método de Jacobi. Matematicamente, temos:

$$\mathrm{x}^{(k+1)} = -(L+D)^{-1}U\mathrm{x}^{(k)} + (L+D)^{-1}\mathrm{b}$$

Utilizando a função **inv()**, a implementação é bastante simples e se assemelha bastante, ao menos do ponto de vista de estrutura, à função que implementa o método de Jacobi. Veja a função a seguir:

```
function [xk, dif_norm, k, residuo] = GaussSeidel_inv(A, b, x0, E, M,
    norma)

    U = triu(A, 1);
    L = tril(A);
    inverse = inv(L)

    xk = x0;
    k = 0;
    dif = E+1;

    while norm((dif), norma)>E & k<M
        xj = xk;
        xk = -inverse*U*xj + inverse*b;
        dif = xk-xj;
        k = k+1;
    end

    dif_norm = norm((dif), norma);
    residuo = norm((b-A*xk), norma);

endfunction
```

A menos do uso da função **inv()** e da reestruturação do modo como encontramos xk (adaptando a equação, obviamente, para o método Gauss-Seidel), não há diferenças notáveis entre esse bloco de código e o anterior, referente ao método de Jacobi. Vamos verificar como implementar o mesmo método de Gauss-Seidel, mas sem utilizar a função **inv()**, custosa do ponto de vista computacional.

2.3 Método de Gauss-Seidel (sem **inv()**)

Desejamos implementar o mesmo método, mas de modo menos custoso. A função abaixo foi estruturada pensando nessa questão:

```
function [xk, dif_norm, k, residuo]=GaussSeidel(A, b, x0, E, M, norma)

    U = triu(A, 1);
    L = tril(A);

    xk = x0;
    xk_size = size(xk, 1);
    k = 0;
    dif = E + 1;

    while k<M & norm(dif, norma)> E
        x = xk;
        U_b = -U*xk + b;
        xk(1) = U_b(1)/L(1, 1);

        for i = 2:xk_size
            xk(i) = (U_b(i) - L(i, 1:i-1)*xk(1:i-1))/L(i, i);
        end

        dif = xk - x
        k = k+1;
    end

    dif_norm = norm((dif), norma);
    residuo = norm((b-A*xk), norma);

endfunction
```

Vamos passar pelo código e verificar o que ele executa:

1. Criamos 2 matrizes: U e L todas a partir da matriz A no *input*. A primeira é triangular superior com zeros na diagonal; a segunda é uma matriz triangular inferior, mas, dessa vez, sua diagonal é a mesma da matriz A;

2. Criamos uma variável xk_size, que, como o nome sugere, guarda o tamanho do vetor xk. Ela será útil posteriormente na iteração. Em seguida, definimos que o vetor xk terá, de início, o valor do vetor x0 do *input*. Além disso, k, que corresponde ao número de iterações, começa em 0 — claro: não há iterações antes do *loop*. A variável dif, redefinida no *loop*, simplesmente é definida como E+1 para que consigamos entrar no *loop*;

3. O *loop* executa o método de Gauss-Seidel, evitando utilizar a função **inv()** do Matlab/Scilab. Executa-se o primeiro caso manualmente e dá-se prosseguimento ao processo utilizando o *loop*. No processo, a diferença entre os vetores xk e o predecessor imediato a ele é calculada a cada iteração, ao mesmo tempo que a variável k é acrescida em uma unidade a cada vez que uma ação do *loop* é executada;

4. Finalmente, criamos as variáveis dif_norm e residuo para que elas guardem, respectivamente, a norma da diferença entre $||x_k - x_{k-1}||$ e o resíduo $||r_k|| = ||b - Ax_k||$.

3 Testagens iniciais

Agora que demonstramos nossas funções e explicamos o funcionamento de cada uma delas, vamos tomar algumas matrizes e definir alguns parâmetros para testá-las. Mas, antes disso, é importante que estejamos atentos para o seguinte teorema que envolve o método de Jacobi: **supondo um sistema $A\mathbf{x} = b$, em que A é quadrada, se A tem diagonal estritamente dominante, então o método de Jacobi converge**. Isto é: conseguimos garantir que esse método — e, por conseguinte, nossa função que o implementa — funciona bem no caso em que tomamos matrizes cuja diagonal é estritamente dominante. Portanto, em um primeiro momento de trabalho, vamos utilizar matrizes dessa natureza para verificar como o método funciona na prática.

3.1 Teste 1

Vejamos um primeiro teste utilizando a matriz $A = \begin{bmatrix} 3 & 1 \\ 1 & 5 \end{bmatrix}$ e o vetor $b = \begin{bmatrix} 9 \\ 17 \end{bmatrix}$. Para esse caso, esperamos um resultado de xk próximo de $x = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$. De início utilizaremos um número máximo de iterações $M = 20$ e uma tolerância de $E = 0.001$. A norma 1 será utilizada e, como de praxe, o vetor inicial será nulo.

Vejamos como cada método se comporta:

```
-->[xk, dif_norm, k, residuo] = Jacobi(A, b, x0, 0.001, 20, 1)
 xk  =
   1.9999605
   2.9999407

 dif_norm  =
   0.0005136

 k  =
   8.

 residuo  =
   0.0005136
```

```
-->[xk, dif_norm, k, residuo] = GaussSeidel_inv(A, b, x0, 0.001, 20, 1)
 xk  =
   2.0000198
   2.999996

```

```
 dif_norm  =
   0.0003319

 k  =
   5.

 residuo  =
   0.0000553
```

```
-->[xk, dif_norm, k, residuo] = GaussSeidel(A, b, x0, 0.001, 20, 1)
 xk  =
   2.0000198
   2.999996

 dif_norm  =
   0.0003319

 k  =
   5.

 residuo  =
   0.0000553
```

De modo geral, podemos fazer algumas comparações entre os dois métodos. Em primeiro lugar, nota-se que ambos os métodos chegaram a aproximações bastante próximas e razoáveis do resultado esperado para o vetor x. Desse ponto de vista, ambos os métodos são eficientes. No entanto, note algumas diferenças: o método de Gauss-Seidel chegou a uma resposta bastante próxima com um número menor de iterações que o método de Jacobi. Ao mesmo tempo, tanto o resíduo quanto a norma da diferença também são menores nesse caso.

Uma observação importante e que impacta na forma como este relatório está estruturado é o fato de que as funções GaussSeidel_inv() e GaussSeidel() retornam os mesmos *outputs*. É claro: trata-se do mesmo método, ainda que implementado de maneiras distintas. As diferenças entre as duas funções não residem nos resultados, mas no tempo de execução, como veremos ao final deste relatório. Para evitar que o relatório fique muito longo e redundante, omitiremos os testes com a GaussSeidel_inv(), já que os resultados são exatamente os mesmos da GaussSeidel(). Prossigamos para mais alguns testes.

3.2 Teste 2

Vejamos um segundo teste utilizando a matriz $A = \begin{bmatrix} 7 & -2 \\ 3 & 8 \end{bmatrix}$ e o vetor $b = \begin{bmatrix} 22 \\ 160 \end{bmatrix}$. Para esse caso, esperamos um resultado de xk próximo de x $= \begin{bmatrix} 8 \\ 17 \end{bmatrix}$. Utilizaremos um número máximo de iterações $M = 20$ e uma tolerância de $E = 0.01$. Dessa vez a norma 2 será utilizada e, como de praxe, o vetor inicial será nulo.

```
-->[xk, dif_norm, k, residuo] = Jacobi(A, b, x0, 0.01, 20, 2)
 xk  =
   7.9989458
   16.99776

 dif_norm  =
   0.0071763

 k  =
   8.

 residuo  =
   0.0212834
```

```
-->[xk, dif_norm, k, residuo] = GaussSeidel(A, b, x0, 0.01, 20, 2)
 xk  =
   7.9993599
   17.00024

 dif_norm  =
   0.0070639

 k  =
   5.

 residuo  =
   0.0049606
```

Mais uma vez, as aproximações dos dois métodos são razoavelmente semelhantes e suficientemente eficientes. Ainda assim, e mais uma vez, o método de GaussSeidel realizou o processo em um número menor de iterações, assim como o resíduo e a norma da diferença seguem sendo menores em relação ao método de Jacobi.

4 Segunda fase de testes

Testaremos os métodos de Jacobi e Gauss-Seidel utilizando a matriz $A = \begin{bmatrix} 1 & -4 & 2 \\ 0 & 2 & 4 \\ 6 & -1 & -2 \end{bmatrix}$, extraída de um sistema de equações sugerido pela lista da Aula Prática 2. Para o vetor b, teremos $b = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$. Nesse caso, o vetor x que resolve o sistema linear é o seguinte: $x = \begin{bmatrix} 1/4 \\ -1/4 \\ 3/8 \end{bmatrix}$. Utilizaremos como x0 um vetor nulo com uma tolerância E = 0.01 e um número máximo de iterações M = 50.

```
-->[xk, dif_norm, k, residuo] = Jacobi(A, b, x0, 0.01, 50, 1)
```

```
 xk  =
  -2.619D+24
  -5.935D+24
   1.117D+25

 dif_norm  =
   2.145D+25

 k  =
   50.

 residuo  =
   1.084D+26
```

```
-->[xk, dif_norm, k, residuo] = GaussSeidel(A, b, x0, 0.01, 50, 1)
 xk  =
  -1.188D+34
  -9.829D+33
  -3.072D+34

 dif_norm  =
   5.838D+34

 k  =
   50.
 residuo  =
   1.765D+35
```

Fica claro que os métodos não funcionaram bem. Para entender o que ocorreu no processo, lembre-se do teorema que enunciamos anteriormente: **supondo um sistema $A\mathbf{x} = b$, em que A é quadrada, se A tem diagonal estritamente dominante, então o método de Jacobi converge**[1]. Note que o processo não funcionou para qualquer um dos métodos e, na prática, isso sugere que o sistema, pelo menos no formato apresentado, não converge para um resultado válido.

Reordenemos as equações do sistema de tal modo que a matriz tenha diagonal estritamente dominante — situação na qual garantimos convergência. Teremos $A = \begin{bmatrix} 6 & -1 & -2 \\ 1 & -4 & 2 \\ 0 & 2 & 4 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ e esperamos que $x = \begin{bmatrix} 1/4 \\ -1/4 \\ 3/8 \end{bmatrix}$. Utilizaremos a mesma tolerância E do processo anterior e definiremos um mesmo número máximo de iterações. Seguiremos utilizando a norma 1.

```
-->[xk, dif_norm, k, residuo] = Jacobi(A, b, x0, 0.01, 50, 1)
 xk  =
```

[1] Isso é importante, mas não podemos perder de vista o fato de que isso **não** quer dizer que o processo só funciona para matrizes com diagonal estritamente dominante.

```
  0.2512418
 -0.2482157
  0.3750362

dif_norm  =
  0.0072097

k  =
  8.

residuo  =
  0.0151307
```

```
-->[xk, dif_norm, k, residuo] = GaussSeidel(A, b, x0, 0.01, 50, 1)
xk  =
  0.25
 -0.2508138
  0.3754069

dif_norm  =
  0.0061035

k  =
  5.

residuo  =
  0.0040690
```

Agora conseguimos encontrar um resultado, em ambos os casos muito próximos do vetor x esperado. O padrão dos testes anteriores se repete: a norma da diferença e o resíduo da Gauss-Seidel são menores que os mesmos valores no método de Jacobi. Além disso, o método também segue mais vantajoso do ponto de vista de número de iterações — acha uma aproximação bastante razoável com menos repetições.

5 Exercício 3 da Lista 2

Tomemos agora a matriz do exercício 3 da lista de exercícios 2 para fazer os testes solicitados com os métodos de Jacobi e Gauss-Seidel. Tome-se $A = \begin{bmatrix} 2 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & 2 \end{bmatrix}$ e $b = \begin{bmatrix} -1 \\ 4 \\ -5 \end{bmatrix}$, para os quais o vetor x esperado será $x = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$.

5.1 Aplicação de Jacobi

Vamos testar o método de Jacobi na norma 1 para 25 iterações e ver como ele se comporta:

```
-->[xk, dif_norm, k, residuo] = Jacobi(A, b, x0, 0.001, 25, 1)
 xk  =
  -20.827873
   2.
  -22.827873

 dif_norm  =
   78.580342

 k  =
   25.

 residuo  =
   174.62298
```

Está claro que a aproximação nem sequer chega perto do vetor x esperado no caso das 25 iterações. Mais do que isso, note que os valores do resíduo e da norma da diferença são altíssimos. Sem dúvida, o método não funcionou bem o suficiente.

5.2 Aplicação de Gauss-Seidel

Agora, nosso objetivo é utilizar o mesmo sistema linear para testar o método de Gauss-Seidel na norma infinito e uma precisão de 10^{-5}:

```
-->[xk, dif_norm, k, residuo] = GaussSeidel(A, b, x0, 10**(-5), 25, %inf
   )
 xk  =
   1.0000023
   1.9999975
  -1.0000001

 dif_norm  =
   0.0000073

 k  =
   23.

 residuo  =
   0.0000069
```

Dessa vez, nosso resultado é eficiente. Com 23 iterações o método chega a um resultado bastante próximo do esperado para o vetor x. Além disso, temos resíduo e norma da diferença baixíssimos, o que sugere que a solução é bastante razoável quando optamos, no método de Gauss Seidel, pela norma infinita e uma tolerância baixa.

6 Exercício 5 da Lista 2

Tomemos agora a matriz do exercício 5 da lista de exercícios 2 para fazer os testes solicitados com o Método de Gauss-Seidel. Tome-se, para isso, a matriz $A = \begin{bmatrix} 1 & 0 & -1 \\ -1/2 & 1 & -1/4 \\ 1 & -1/2 & 1 \end{bmatrix}$ e o vetor $b = \begin{bmatrix} 0.2 \\ -1.425 \\ 2 \end{bmatrix}$, para os quais o x esperado que soluciona o sistema é $x = \begin{bmatrix} 0.9 \\ -0.8 \\ 0.7 \end{bmatrix}$.

6.1 Primeira aplicação

No primeiro caso, vamos tomar um máximo de 300 iterações e uma tolerância de 10^{-2}:

```
-->[xk, dif_norm, k, residuo] = GaussSeidel(A, b, x0, 10**(-2), 300, 1)
 xk  =
   0.9015543
  -0.7988343
   0.6990286

 dif_norm  =
   0.0095979

 k  =
   14.

 residuo  =
   0.0031572
```

Nosso resultado nesse teste é bastante próximo do esperado. A norma da diferença e o resíduo são pequenos e conseguimos chegar a um resultado bastante razoável com apenas 14 iterações — um número bastante menor que o nosso máximo definido de 300 iterações.

6.2 Aplicação após alteração no sistema

Vamos seguir para uma segunda testagem utilizando o mesmo método, mas fazendo uma pequena alteração na matriz, que agora será $A = \begin{bmatrix} 1 & 0 & -2 \\ -1/2 & 1 & -1/4 \\ 1 & -1/2 & 1 \end{bmatrix}$. O vetor b segue sendo exatamente o mesmo. Verifique o teste:

```
-->[xk, dif_norm, k, residuo] = GaussSeidel(A, b, x0, 10**(-2), 300, 1)
 xk  =
   2.157D+41
   1.348D+41
  -1.483D+41

```

```
 dif_norm  =
   8.615D+41

 k  =
   300.

 residuo  =
   5.763D+41
```

Note que uma simples alteração na matriz fez com que o sistema deixasse de convergir. Os resultados apresentados são pouco razoáveis, com resíduo e norma da diferença altos. Note, inclusive, que o número máximo de iterações foi alcançado, sem que um resultado suficientemente razoável fosse retornado.

7 Avaliação dos tempos de execução

Vimos anteriormente que as funções GaussSeidel() e GaussSeidel_inv() implementam o mesmo método iterativo — o método de Gauss-Seidel, evidentemente, mas de maneiras distintas do ponto de vista de código. A função GaussSeidel_inv() tem implementação bastante simples, mas utiliza a função inv() do Matlab/Scilab, cuja função é fazer o cálculo da inversa. No entanto, a execução desse processo é extremamente custosa e lenta computacionalmente. Façamos alguns testes com matrizes genéricas geradas aleatoriamente, tomando o cuidado de que sejam matrizes com diagonal estritamente dominante.

Tomemos, por exemplo, uma matriz 10×10, gerada com valores aleatórios:

```
--> D = diag(fix(rand(10, 1, "uniform")*1000));
--> R = fix(rand(10, 10, "uniform")*10);
--> A = D + R;

--> x = fix(rand(10, 1, "uniform")*10);
--> b = A*x;

--> x0 = zeros(10, 1);
```

Vejamos o que esse código executa:

1. Primeiro, criamos uma matriz diagonal D com distribuição uniforme nos valores da diagonal, todos inteiros. Para garantir que esses valores serão suficientemente grandes, multiplicamos cada um deles por 1000;

2. Depois, criamos uma matriz R que corresponde aos demais valores que comporão a matriz A criada em seguida;

3. Criamos uma matriz A, resultante da soma de D e R;

4. Em seguida, criamos um vetor x qualquer (aleatório) e buscamos por um b resultante da multiplicação de A por x;

5. Finalmente, criamos um vetor nulo, que será nosso x0, do tamanho da matriz em questão.

Perceba que, ao longo do relatório, faremos testes para matrizes maiores que 10 por 10. Para isso, evidentemente que o código será adaptado conforme as necessidades de cada caso. Dentre eles, o ajuste do tamanho da matriz é o mais trivial, mas há casos nos quais será necessário aumentar os valores das entradas, multiplicando por números maiores, para que as execuções do método ocorram de forma correta.

Verifique a seguir o teste do método de Gauss-Seidel para uma matriz 10 por 10 aleatória:

```
--> [xk, dif_norm, k, residuo]=GaussSeidel(A, b, x0, 0.01, 1000, 1)
 xk  =
   6.9997866
  -0.0000449
   5.0038211
   2.9999345
   1.9999830
   5.9999483
   1.0000052
   5.9999811
   5.9999968
   2.9998305

 dif_norm  =
   0.0070870

 k  =
   11.

 residuo  =
   0.0606818
```

De fato, funciona bem. Mas e o tempo de execução? Veja a seguir:

7.1 Matriz 10 × 10

```
--> tic
--> GaussSeidel_inv(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   0.0197303
```

```
--> tic
--> GaussSeidel(A, b, x0, 0.01, 1000, 1);
--> toc
```

```
 ans  =
   0.0204935
```

Perceba que, para matrizes 10 por 10, não há grandes diferenças no tempo de execução para os dois métodos. Aliás, a função que calcula diretamente a inversa acaba sendo um pouco mais rápida.

7.2 Matriz 100 × 100

```
--> tic
--> GaussSeidel_inv(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   0.0234282
```

```
--> tic
--> GaussSeidel(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   0.0275418
```

No caso de matrizes 100 por 100, o padrão se repete. As diferenças no tempo de execução ainda não demonstram vantagem para qualquer uma das funções, porque encontra-se uma solução para o sistema em tempo razoavelmente semelhante.

7.3 Matriz 1000 × 1000

```
--> tic
--> GaussSeidel_inv(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   0.631168
```

```
--> tic
--> GaussSeidel(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   0.6342699
```

Aparentemente, matrizes 1000 por 1000 ainda não são grandes o suficiente para deixarem evidentes as diferenças entre as duas funções. Os tempos de execução são muito próximos, com uma leve vantagem para a função que faz o cálculo da inversa. Vejamos até quando esse padrão se mantém.

7.4 Matriz 2000 × 2000

```
--> tic
--> GaussSeidel_inv(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   15.418107
```

```
--> tic
--> GaussSeidel(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   8.5944372
```

Agora percebemos outro comportamento. Com uma matriz 2000 por 2000, o tempo de execução da função GaussSeidel_inv(), que faz o cálculo da inversa, é razoavelmente maior do que o tempo de execução da GaussSeidel(). É verdade que não é uma diferença muito significativa — afinal, trata-se de uma diferença de pouco menos de 7 segundos. Mesmo assim, já demonstra que um método é mais eficaz que o outro (do ponto de vista computacional) quando o tamanho da matriz aumenta.

7.5 Matriz 5000 $\times$ 5000

```
--> tic
--> GaussSeidel_inv(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   2566.0805
```

```
--> tic
--> GaussSeidel(A, b, x0, 0.01, 1000, 1);
--> toc
 ans  =
   152.91472
```

No caso de uma matriz ainda maior, de 5000 por 5000, a diferença fica muito clara. O tempo de execução da GaussSeidel() é muito menor que o tempo de execução da GaussSeidel_inv(). De modo geral, o que vale notar é que, quanto maior o tamanho da matriz, maior é o custo computacional de se utilizar funções que calculam a inversa — isto é, utilizando a função inv(). No mundo da Ciência de Dados, em que os trabalhos são complexos e lidam com matrizes gigantescas, fica clara a importância de se desenvolverem métodos alternativos que "driblem"o cálculo da inversa de matrizes e, por consequência, sejam executados mais velozmente.