Iniciaremos definindo duas matrizes, as quais serão usadas nos primeiros testes do algoritmos propostos. Vamos definir uma matriz 5x5 qualquer, preenchida por inteiros (A = [1, 2, 3, 4, 5; 0, 6, 7, 8, 9; 9, 0, 10, 11, 12; 0, 1, 0, 13, 14; 0, 7, 0, 2, 15]) e uma matriz 3x3, também preenchida por inteiros (B = [1, 2, 3; 0, 4, 5; 8, 1, 6]).

```
A  =

   1.    2.    3.    4.    5.
   0.    6.    7.    8.    9.
   9.    0.    10.   11.   12.
   0.    1.    0.    13.   14.
   0.    7.    0.    2.    15.
```

```
B  =

   1.    2.    3.
   0.    4.    5.
   8.    1.    6.
```

Não vamos usar matrizes cuja instabilidade numérica é mais provável, pois iremos realizar esse teste posteriormente.

Questão 1 -

Aqui, implementamos o algoritmo de Gram-Schmidt teórico, sem se preocupar com as instabilidades numéricas computacionais. A implementação para esse algoritmo em scilab ficou da seguinte forma:

```
function [Q, R] = qr_GS(A)
    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);

    for j = 1:n
        coluna = A(:, j); //jº coluna de A

        //seleciona todas colunas já ortogonormalizadas
        for i = 1:j-1
            R(i, j) = Q(:, i)' * A(:, j);
            //gram schmidt, v_i já está normalizado
            coluna = coluna - R(i, j) * Q(:, i);
        end
        R(j, j) = norm(coluna, 2);
        Q(:, j) = coluna/R(j, j); //normaliza a coluna ortogonal resultante
    end
endfunction
```

Vemos aqui, que a matriz R é definida durante as iterações, para que não haja a necessidade de realizar $R = Q^T A$, já que isso pode ser custoso computacionalmente, se utilizarmos matrizes muito grandes.

Bom, começamos os testes obtendo a matriz QR de A e analisando $Q^T Q$ e $QR - A$, onde a primeira deve o mais próximo possível da identidade, já que $Q^T = Q^{-1}$ e a segunda deve ser igual a uma matriz de zeros, já que $QR = A$.

```
--> [Q, R] = qr_GS(A)
 Q  =

   0.1104315   0.208304    0.1592151   0.0301666  -0.958204
   0.          0.632627    0.729974   -0.0081402   0.258563
   0.9938837  -0.0231449  -0.0176906  -0.0033518   0.1064671
   0.          0.1054378  -0.0939655   0.9892294   0.0384512
   0.          0.7380648  -0.6577584  -0.1429602   0.0466541
 R  =

   9.0553851   0.2208631   10.270132   11.374447   12.478762
   0.          9.4842617   4.8218521   8.4864597   19.004527
   0.          0.          5.4105575   3.744988   -4.0283378
   0.          0.          0.          12.592737   11.742158
   0.          0.          0.          0.          0.051781

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.          3.469D-18   3.712D-16   1.826D-16   2.191D-14
   3.469D-18   1.          5.551D-17   9.714D-17   8.522D-14
   3.712D-16   5.551D-17   1.         -4.441D-16  -4.376D-14
   1.826D-16   9.714D-17  -4.441D-16   1.         -9.637D-14
   2.191D-14   8.524D-14  -4.377D-14  -9.637D-14   1.

--> disp(Q * R - A);  // Deve ser aproximadamente a matriz zero

   0.   0.          0.         -4.441D-16  -8.882D-16
   0.   0.          0.          0.         -1.776D-15
   0.  -2.776D-17   0.          0.          0.
   0.   0.          0.          0.          1.776D-15
   0.   0.          4.441D-16   4.441D-16   0.
```

Aparentemente, encontramos um resultado satisfatório. No entanto, podemos ver que ainda há espaço para melhorias, pois em várias posições, há a presença de números muito pequenos, mas não zero. Podemos usar a norma de Frobenius para calcular o quão perto essas matrizes estão do resultado desejado, uma vez que para a matriz zero, a norma de Frobenius é zero. Então, fazemos:

```
--> norm(Q * R - A, "fro")
 ans  =

   2.773D-15

--> norm(Q'*Q - eye(5,5), "fro")
 ans  =

   1.947D-13
```

Vemos aqui os valores **1,9E-13** e **2,7E-15**. Agora, realizamos o mesmo teste para a matriz B:

```
--> [Q, R] = qr_GS(B)
 Q  =

   0.1240347   0.4184844  -0.8997145
   0.          0.9067163   0.4217412
   0.9922779  -0.0523106   0.1124643
 R  =

   8.0622577   1.2403473   6.3257715
   0.          4.4115234   5.4751713
   0.          0.          0.0843482

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.          -1.388D-17  -1.958D-14
  -1.388D-17    1.         -1.994D-14
  -1.958D-14   -1.994D-14   1.

--> disp(Q * R - B);  // Deve ser aproximadamente a matriz zero

   0.   0.   0.
   0.   0.   0.
   0.   0.   0.
```

Incrivelmente, tivemos um resultado muito bom, mas ainda com algum erro. Vamos também calcular a norma de Frobenius aqui:

```
--> norm(Q'*Q - eye(3,3), "fro")
 ans  =

   3.953D-14

--> norm(Q * R - B, "fro")
 ans  =

   0.
```

Vemos aqui os valores **3,9E-14** e **0**.

Questão 2 -

Aqui, implementamos o algoritmo de Gram-Schmidt modificado, levando em conta as possíveis instabilidades numéricas computacionais. A implementação para esse algoritmo em scilab ficou da seguinte forma:

```
function [Q, R] = qr_GSM(A)
    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);

    for j = 1:n
        coluna = A(:, j); //jº coluna de A

        //seleciona todas colunas já ortogonormalizadas
        for i = 1:j-1
            R(i, j) = Q(:, i)' * coluna;
            //gram schmidt, v_i já está normalizado
            coluna = coluna - R(i, j) * Q(:, i);
        end
        R(j, j) = norm(coluna, 2);
        Q(:, j) = coluna/R(j, j); //normaliza a coluna ortogonal resultante
    end
endfunction
```

A alteração em relação ao algoritmo anterior está nessa linha:

```
R(i, j) = Q(:, i)' * coluna;
```

Aqui, ao invés de projetar a coluna $A(:, j)$ em todas as iterações, projetamos o vetor já projetado nos vetores anteriores. Isso garante que estamos sempre trabalhando com vetores independentes e ortogonais, diminuindo também a instabilidade numérica, resultando em uma decomposição QR mais eficiente e estável.

Assim, podemos partir para o teste com as matrizes A e B:

```
--> [Q, R] = qr_GSM(A)
 Q  =

   0.1104315   0.208304    0.1592151   0.0301666  -0.958204
   0.          0.632627    0.729974   -0.0081402   0.258563
   0.9938837  -0.0231449  -0.0176906  -0.0033518   0.1064671
   0.          0.1054378  -0.0939655   0.9892294   0.0384512
   0.          0.7380648  -0.6577584  -0.1429602   0.0466541
 R  =

   9.0553851   0.2208631   10.270132   11.374447   12.478762
   0.          9.4842617   4.8218521   8.4864597   19.004527
   0.          0.          5.4105575   3.744988   -4.0283378
   0.          0.          0.          12.592737   11.742158
   0.          0.          0.          0.          0.051781

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.          3.469D-18   3.712D-16   1.830D-16   2.198D-14
   3.469D-18   1.          2.220D-16   5.551D-17   1.062D-13
   3.712D-16   2.220D-16   1.         -5.551D-17   2.583D-14
   1.830D-16   5.551D-17  -5.551D-17   1.          8.145D-14
   2.198D-14   1.062D-13   2.584D-14   8.145D-14   1.

--> disp(Q * R - A);  // Deve ser aproximadamente a matriz zero

   0.   0.          0.   0.          0.
   0.   0.          0.   0.          0.
   0.  -2.776D-17   0.   0.          1.776D-15
   0.   0.          0.   0.          0.
   0.   0.          0.   4.441D-16   0.
```

Os resultados são satisfatórios. Agora, calculamos a norma para comparar com o algoritmo anterior:

```
--> norm(Q'*Q - eye(5,5), "fro")
 ans  =

   1.953D-13

--> norm(Q * R - A, "fro")
 ans  =

   1.831D-15
```

Vemos aqui, os valores **1,9E-13** e **1,83E-15B**, que é um resultado melhor do que o anterior. Agora, realizando o teste para a matriz B:

```
--> [Q, R] = qr_GSM(B)
 Q  =

   0.1240347   0.4184844  -0.8997145
   0.          0.9067163   0.4217412
   0.9922779  -0.0523106   0.1124643
 R  =

   8.0622577   1.2403473   6.3257715
   0.          4.4115234   5.4751713
   0.          0.          0.0843482

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.         -1.388D-17  -1.958D-14
  -1.388D-17   1.         -1.994D-14
  -1.958D-14  -1.994D-14   1.

--> disp(Q * R - B);  // Deve ser aproximadamente a matriz zero

   0.   0.   0.
   0.   0.   0.
   0.   0.   0.
```

Os resultados são satisfatórios. Agora, calculamos a norma para comparar com o algoritmo anterior:

```
--> norm(Q'*Q - eye(3,3), "fro")
 ans  =

   3.953D-14

--> norm(Q * R - B, "fro")
 ans  =

   0.
```

Vemos aqui, os valores **3,9E-14** e **0**, que é um resultado igual ao anterior. Nesse caso, não houve nenhuma diferença, pois não houve nenhuma instabilidade numérica durantes os cálculos para essa matriz.

Questão 3 -

Aqui, implementamos o algoritmo de Gram-Schmidt modificado com pivoteamento de colunas. A implementação para esse algoritmo em scilab ficou da seguinte forma:

```
function [Q, R, P] = qr_GSP(A)
    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);
    P = eye(n, n);

    for j = 1:n
        //verifica a coluna de maior norma
        norms = zeros(1, n-j+1);

        for k = j:n //calcula a norma de todas as colunas
            norms(k-j+1) = norm(A(:, k));
        end
        [value, idx] = max(norms); //verifica o indice da coluna de maior norma
        idx = idx + j - 1;

        if idx ~= j //permuta as colunas j e idx
            A(:, [j, idx]) = A(:, [idx, j]);
            P(:, [j, idx]) = P(:, [idx, j]);
        end

        //gram schmidt modificado
        coluna = A(:, j); //jº coluna de A

        //seleciona todas colunas já ortogonormalizadas
        for i = 1:j-1
            R(i, j) = Q(:, i)' * coluna;
            //gram schmidt, v_i já está normalizado
            coluna = coluna - R(i, j) * Q(:, i);
        end
        R(j, j) = norm(coluna, 2);
        Q(:, j) = coluna/R(j, j); //normaliza a coluna ortogonal resultante
    end
endfunction
```

Aqui, a diferença é que a cada iteração, com exceção da primeira, verificamos a norma das colunas projeções delas sobre o subespaço gerado pela última coluna ortonormal obtida, então escolhemos a coluna de maior norma. Na primeira iteração, apenas escolhemos a de maior norma.

Assim, podemos partir para o teste com as matrizes A e B:

```
--> [Q, R, P] = qr_GSP(A)
 Q  =

   0.1930229   0.0788697   0.1308553   0.1502378   0.957511
   0.3474411   0.2259681   0.3721188   0.7877345  -0.2631065
   0.4632549   0.3362918   0.5530665  -0.5964855  -0.1030787
   0.540464    0.409841   -0.7334208  -0.0224931  -0.0389496
   0.5790686  -0.8134225  -0.0248169  -0.024538   -0.0424905
 R  =

   25.903668   16.831593   7.6437052   7.0646366   4.3623166
   0.          9.5235217   5.1813041  -3.7705685   3.1054962
   0.          0.          8.528063    1.5872845   5.108454
   0.          0.          0.          4.8326236  -5.2181315
   0.          0.          0.          0.          0.0298023
 P  =

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

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.          1.110D-16  -1.266D-16   1.370D-16   4.883D-14
   1.110D-16   1.          1.110D-16  -1.006D-16  -3.125D-14
  -1.266D-16   1.110D-16   1.          1.160D-16   7.218D-14
   1.370D-16  -1.006D-16   1.160D-16   1.          1.221D-14
   4.882D-14  -3.125D-14   7.218D-14   1.221D-14   1.

--> disp(Q * R - A*P);  // Deve ser aproximadamente a matriz zero

   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.
```

Os resultados são muito bons. Pela primeira vez, conseguimos zerar $QR - AP$. Aqui, vale mencionar que precisamos de AP, pois com o pivoteamento a decomposição ficou $AP = QR$. Agora, calculamos a norma para comparar com o algoritmo anterior:

```
--> norm(Q'*Q - eye(5,5), "fro")
 ans  =

   1.321D-13

--> norm(Q * R - A*P, "fro")
 ans  =

   0.
```

Aqui, concluímos que realmente tivemos resultados melhores do que nos dois outros algoritmos testados. Agora, vamos passar para a matriz B:

```
--> [Q, R, P] = qr_GSP(B)
 Q  =

   0.3585686  -0.2247104   0.906054
   0.5976143  -0.6903753  -0.4077243
   0.7171372   0.6876679  -0.1132567
 R  =

   8.3666003   6.0956659   3.8247315
   0.          5.2766331  -2.523254
   0.          0.          0.067954
 P  =

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

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.          5.551D-16   4.706D-14
   5.551D-16   1.          1.679D-15
   4.706D-14   1.679D-15   1.

--> disp(Q * R - B*P);  // Deve ser aproximadamente a matriz zero

   0.   0.   0.
   0.   0.   0.
   0.   0.   0.
```

Satisfatório, mas não podemos concluir nada de cara, então calcularemos a norma:

```
--> norm(Q'*Q - eye(3,3), "fro")
 ans  =

   6.660D-14

--> norm(Q * R - B*P, "fro")
 ans  =

   0.
```

Bom, agora sim, podemos comparar. Contudo, aqui a norma está pior que nos algoritmos anteriores, diferente do que era esperado. Contudo, é uma diferença insignificante.

Assim, concluímos que em termos de precisão e instabilidade, podemos rankear os algoritmos da seguinte forma:

$$Modificado\ com\ pivoteamento\ >\ Modificado\ >\ Teórico$$

Questão 4 -

Aqui, começamos apresentando as duas implementações:

```
function [U, R] = qr_house_v1(A)
    [m, n] = size(A);
    k = min(m,n); //define sobre quem iterar
    U = zeros(m, n);
    R = A;

    for j = 1:k //se j > m ou j > n para
        x = R(j:m, j); //elementos abaixo da diagonal

        e = zeros(length(x), 1);
        e(1) = norm(x);

        //para evitar erros númericos
        if x(1) > 0
            u = x + e;
        else
            u = x - e;
        end

        u = u / norm(u); //normaliza u

        //transformação de Householder
        R(j:m, j:n) = R(j:m, j:n) - 2 * u * (u' * R(j:m, j:n));
        U(j:m, j) = u;
    end
endfunction
```

Vale mencionar, que aqui resolvi iterar apenas até o **min(m, n),** porque se $j > m$ ou $j > n$, $R(j{:}m, j{:}n)$ não existirá. Então, como essas interações são irrelevantes, resolvi iterar apenas sob a diagonal da matriz e definir os vetores restantes como 0, formando a matriz da ordem requerida. Assim, para a segunda versão, basta alterarmos o código para iterar sobre **min(m-1, n):**

```
function [U, R] = qr_house_v2(A)
    [m, n] = size(A);
    k = min(m-1, n); //define sobre quem iterar (m-1)
    U = zeros(m, k);
    R = A;

    for j = 1:k //se j > m-1 ou j > n para
        x = R(j:m, j); //elementos abaixo da diagonal

        e = zeros(length(x), 1);
        e(1) = norm(x);

        //para evitar erros númericos
        if x(1) > 0
            u = x + e;
        else
            u = x - e;
        end

        u = u / norm(u); //normaliza u

        //transformação de Householder
        R(j:m, j:n) = R(j:m, j:n) - 2 * u * (u' * R(j:m, j:n));
        U(j:m, j) = u;
    end
endfunction
```

Portanto, basta nós criarmos a função que acha a matriz Q a partir da U resultante. Fazemos isso da seguinte forma:

```
function Q = constroi_Q_House(U)
    [m, k] = size(U);
    Q = eye(m, m);

    //percorre do fim para o começo da matriz
    for j = k:-1:1
        u = U(:, j); //seleciona a coluna

        //atualiza Q aplicando a transformação de Householder
        Q = Q - 2 * u * (u' * Q);
    end
endfunction
```

Dessa forma, podemos achar a decomposição QR das matrizes. Então vamos testar esses algoritmos nas mesmas matrizes usadas anteriormente. Começamos com a matriz A.

```
--> [U, R] = qr_house_v1(A);

--> Q = constroi_Q_House(U);

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.          5.204D-17  -2.776D-17   7.806D-18   1.527D-16
   5.204D-17   1.         -1.110D-16   1.388D-17   1.388D-17
  -2.776D-17  -1.110D-16   1.          2.776D-17   4.441D-16
   7.806D-18   1.388D-17   2.776D-17   1.         -9.281D-17
   1.527D-16   0.          4.441D-16  -9.714D-17   1.

--> disp(Q * R - A);  // Deve ser aproximadamente a matriz zero

   1.776D-15   8.882D-16   4.441D-16  -4.441D-16   7.105D-15
   0.          1.776D-15  -8.882D-16   1.776D-15   3.553D-15
   1.776D-15   4.441D-16   1.776D-15   3.553D-15   3.553D-15
   0.          2.220D-16   4.087D-16  -1.599D-14  -1.066D-14
   0.          0.          2.665D-15   4.441D-15   0.
```

Um resultado promissor, mas da mesma forma que nos outros algoritmos, vamos analisar a norma:

```
--> norm(Q'*Q - eye(5,5), "fro")
 ans  =

   1.662D-15

--> norm(Q * R - A, "fro")
 ans  =

   2.242D-14
```

Vemos aqui, os valores **1,6E-15** e **2,2E-14.** A primeira métrica foi a melhor comparativamente a todos os algoritmos, contudo a segunda norma ficou um pouco pior que a obtida pelos outros algoritmos. Agora, partiremos para o teste com a matriz B:

```
--> [U, R] = qr_house_v1(B);

--> Q = constroi_Q_House(U);

--> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

   1.          2.220D-16  -5.135D-16
   2.220D-16   1.         -2.299D-16
  -5.135D-16  -2.299D-16   1.

--> disp(Q * R - B);  // Deve ser aproximadamente a matriz zero

   1.332D-15   1.776D-15   3.109D-15
   7.492D-16   0.          0.
   5.329D-15   1.554D-15   3.553D-15
```

Notamos uma acurácia um pouco inferior àquela encontrada com o método de Gram-Schmidt. Analisando a norma, temos:

```
--> norm(Q'*Q - eye(3,3), "fro")
 ans  =

   9.890D-16

--> norm(Q * R - B, "fro")
 ans  =

   7.655D-15
```

Agora, percebemos que o método obteve uma precisão superior aos seus pares que utilizam Gram-Schmidt, apesar de obter uma acurácia inferior. Para comparação, usaremos a segunda versão para os mesmos testes. Aqui vamos apresentar apenas as normas:

```
-> [U, R] = qr_house_v2(A);

-> Q = constroi_Q_House(U);

-> norm(Q'*Q - eye(5,5), "fro")
ans  =

  1.662D-15

-> norm(Q * R - A, "fro")
ans  =

  2.249D-14
```

```
-> [U, R] = qr_house_v2(B);

-> Q = constroi_Q_House(U);

-> norm(Q'*Q - eye(3,3), "fro")
ans  =

  9.890D-16

-> norm(Q * R - B, "fro")
ans  =

  8.749D-15
```

Assim, percebemos que usando essa segunda versão (1) para a matriz A não houve diferença alguma no resultado, (2) para a matriz B, apenas houve um aumento (**1,1E-15**) insignificante na norma da acurácia. Assim, podemos concluir que o uso de ambas as versões é similar em termos de resultado, contudo, vale analisar os aspectos computacionais de cada um.

Na primeira versão, estamos armazenando todos os vetores unitários que geram as matrizes dos refletores de Householder. Por outro lado, na segunda versão, estamos mantendo apenas os vetores unitários necessários para refletir as colunas de A. Como os vetores unitários de Householder são usados para zerar os elementos abaixo da diagonal principal, apenas $min(m - 1, n)$ vetores unitários são necessários, pois não precisamos zerar os elementos acima da diagonal e, consequentemente, as colunas cujo não armazenam nenhum pivô. Isso resulta em uma matriz U com menos colunas do que A, pois estamos armazenando apenas o necessário para a decomposição QR.

Portanto, a principal diferença entre essas duas versões está na quantidade de memória necessária para armazenar a matriz U e na eficiência computacional, pois na segunda versão, estamos economizando espaço armazenando apenas os vetores unitários necessários para a decomposição QR. Isso pode ser vantajoso especialmente quando lidamos com matrizes grandes, pois reduzimos a quantidade de cálculos e o uso de memória. Podemos observar a diferença na matriz U de cada algoritmo, como apresentado abaixo:

```
--> U = qr_house_v2(A)
 U  =

   0.745128    0.          0.          0.
   0.          0.9035007   0.          0.
   0.6669215  -0.1159855  -0.7302348   0.
   0.          0.0583496  -0.0966185   0.9984762
   0.          0.4084473  -0.6763298  -0.0551845

--> U = qr_house_v1(A)
 U  =

   0.745128    0.          0.          0.          0.
   0.          0.9035007   0.          0.          0.
   0.6669215  -0.1159855  -0.7302348   0.          0.
   0.          0.0583496  -0.0966185   0.9984762   0.
   0.          0.4084473  -0.6763298  -0.0551845  -1.
```

```
-> [U, R] = qr_house_v2(A);

-> Q = constroi_Q_House(U);

-> disp(Q' * Q);  // Deve ser aproximadamente a matriz identidade

  1.          5.204D-17  -2.776D-17   7.806D-18  -1.527D-16
  5.204D-17   1.         -1.110D-16   1.388D-17  -1.388D-17
 -2.776D-17  -1.110D-16   1.          2.776D-17  -4.441D-16
  7.806D-18   1.388D-17   2.776D-17   1.          9.281D-17
 -1.527D-16   0.         -4.441D-16   9.714D-17   1.

-> disp(Q * R - A);  // Deve ser aproximadamente a matriz zero

  1.776D-15   8.882D-16  -1.332D-15  -1.332D-15   7.105D-15
  0.          1.776D-15  -8.882D-16   1.776D-15   3.553D-15
  1.776D-15   4.441D-16   1.776D-15   3.553D-15   3.553D-15
  0.          2.220D-16   4.770D-16  -1.599D-14  -1.066D-14
  0.          0.          2.665D-15   4.441D-15   0.



-> norm(Q'*Q - eye(5,5), "fro")
ans  =

  1.662D-15

-> norm(Q * R - A, "fro")
ans  =

  2.249D-14
```

Questão 4.2 -

Aqui, vamos comparar os algoritmos criados usando as matrizes:

- M1 = testmatrix('magi', 7);
- H = testmatrix('hilb', 7);
- M2 = testmatrix('magi', 6);

Para não se estender muito, iremos apresentar aqui apenas as normas dos resultados, comprando-as. Começaremos com a matriz M1:

```
-> [Q, R] = qr_GS(M1);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  1.993D-15

-> norm(Q * R - M1, "fro")
ans  =

  1.476D-14

-> [Q, R] = qr_GSM(M1);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  1.097D-15

-> norm(Q * R - M1, "fro")
ans  =

  1.081D-14
```

```
-> [U, R] = qr_house_v2(M1);

-> Q = constroi_Q_House(U);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  2.220D-15

-> norm(Q * R - M1, "fro")
ans  =

  1.411D-13

-> [Q, R] = qr(M1);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  9.431D-16

-> norm(Q * R - M1, "fro")
ans  =

  5.013D-14
```

Aqui, podemos ver que todas tiveram um bom resultado. Contudo, a melhor implementação é a nativa do Scilab, por uma pequena margem, já que o método de Gram-Schmidt modificado chegou realmente muito perto. Em comparação, eu colocaria a performance deles:

$$qr > gsm > gs > house$$

Testando agora para a matriz H, temos:

```
-> [Q, R] = qr_GS(H);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  1.2796432

-> norm(Q * R - H, "fro")
ans  =

  2.687D-08

-> [Q, R] = qr_GSM(H);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  9.158D-09

-> norm(Q * R - H, "fro")
ans  =

  1.936D-08
```

```
-> [U, R] = qr_house_v2(H);

-> Q = constroi_Q_House(U);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  1.685D-15

-> norm(Q * R - H, "fro")
ans  =

  0.0000001

-> [Q, R] = qr(H);

-> norm(Q'*Q - eye(7,7), "fro")
ans  =

  1.119D-15

-> norm(Q * R - H, "fro")
ans  =

  5.671D-08
```

Nessa análise, observamos que o método de Gram-Schmidt padrão apresentou problemas, já que sua norma de precisão foi de **1,2**, um número muito alto, indicando que ele não convergiu para a matriz desejada. Por outro lado, o Gram-Schmidt modificado e o Householder tiveram um bom desempenho, sendo difícil classificar qual teve a melhor performance. Contudo, eu classificaria da seguinte forma:

$$qr > house > gsm >> gs$$

Esse resultado do método de Gram-Schmidt padrão se deve à sua instabilidade. Isso pode levar a imprecisões na matriz Q, que, nesse caso, foram grandes o suficiente para serem perceptíveis. Testando agora para a matriz H, temos:

```
--> [Q, R] = qr_GS(M2);

--> norm(Q'*Q - eye(6,6), "fro")
 ans  =

   1.4141521

--> norm(Q * R - M2, "fro")
 ans  =

   8.882D-15

--> [Q, R] = qr_GSM(M2);

--> norm(Q'*Q - eye(6,6), "fro")
 ans  =

   1.4092295

--> norm(Q * R - M2, "fro")
 ans  =

   1.390D-14
```

```
--> [U, R] = qr_house_v2(M2);

--> Q = constroi_Q_House(U);

--> norm(Q'*Q - eye(6,6), "fro")
 ans  =

   1.240D-15

--> norm(Q * R - M2, "fro")
 ans  =

   3.243D-14

--> [Q, R] = qr(M2);

--> norm(Q'*Q - eye(6,6), "fro")
 ans  =

   1.071D-15

--> norm(Q * R - M2, "fro")
 ans  =

   3.266D-14
```

Bom, agora tivemos problema em ambos os métodos de Gram-Schmidt, então resolvi executar a versão com pivoteamento:

```
--> [Q, R, P] = qr_GSP(M2);

--> norm(Q'*Q - eye(6,6), "fro")
 ans  =

   1.4140384

--> norm(Q * R - M2*P, "fro")
 ans  =

   7.536D-15
```

Que também não obteve um bom resultado, pintando as matrizes R e Q com relação ao método qr(), vemos:

```
--> [Q, R] = qr_GSM(M2);

--> Q, R
 Q  =

   0.6211496  -0.1702323  -0.2069923   0.4998127  -0.2062416   0.6500795
   0.0532414   0.5740207  -0.4500167   0.2106392   0.6486769  -0.2935843
   0.550161   -0.0011211  -0.4460032  -0.4537259  -0.2062416   0.4194061
   0.141977    0.4732929   0.3762869   0.503413   -0.3328962  -0.2306734
   0.5324139  -0.0695045   0.628716   -0.2095554   0.5220223   0.4194061
   0.0709885   0.642404    0.1372761  -0.4501256  -0.3328962  -0.2935843
 R  =

   56.347138   16.469337   30.045891   39.096928   38.0321     38.670997
   0.          54.219562   34.879737   23.166906   25.260929   23.296284
   0.          0.          32.490742  -8.9181606  -11.28946   -7.9244985
   0.          0.          0.          7.6283088  -3.911361    7.4338955
   0.          0.          0.          0.          3.4196741   6.8393482
   0.          0.          0.          0.          0.          2.118D-14

--> [Q_sci, R_sci] = qr(M2)
 Q_sci  =

  -0.6211496   0.1702323  -0.2069923  -0.4998127   0.2062416  -0.5
  -0.0532414  -0.5740207  -0.4500167  -0.2106392  -0.6486769   1.114D-15
  -0.550161    0.0011211  -0.4460032   0.4537259   0.2062416   0.5
  -0.141977   -0.4732929   0.3762869  -0.503413    0.3328962   0.5
  -0.5324139   0.0695045   0.628716    0.2095554  -0.5220223   4.718D-16
  -0.0709885  -0.642404    0.1372761   0.4501256   0.3328962  -0.5
 R_sci  =

  -56.347138  -16.469337  -30.045891  -39.096928  -38.0321    -38.670997
   0.         -54.219562  -34.879737  -23.166906  -25.260929  -23.296284
   0.          0.          32.490742  -8.9181606  -11.28946   -7.9244985
   0.          0.          0.         -7.6283088   3.911361   -7.4338955
   0.          0.          0.          0.         -3.4196741  -6.8393482
   0.          0.          0.          0.          0.          1.688D-14
```

Assim, podemos concluir que o problema está na última coluna de Q e no último pivô de R. Não consegui inferir exatamente o motivo do problema, mas possivelmente acontece pelos números pequenos (muito próximos de 0) apresentados na matriz correta. Possivelmente está havendo um erro no momento dos cálculos. Por isso, após as análises, considerando a estabilidade e precisão dos métodos, eu classificaria eles da seguinte forma:

$$qr > householder > GS\_modificado >> GS\_padrão$$

Questão 5 -

Inicialmente, elaborei a função que calcula os autovalores de uma matriz simétrica A, a partir da decomposição QR. Vale mencionar, que aqui há a necessidade da escolha do algoritmo que será usado para achar a decomposição. Aqui nesta atividade prática, já construímos vários algoritmos e observamos os seus comportamentos. Então, para não focar nisso novamente, resolvi usar o método nativo do scilab para achar essa decomposição. Mas é claro que, usando qualquer um dos outros algoritmos, essa função funcionaria da mesma forma.

```
function S = espectro(A, tol)
    n = size(A, 1);
    S = diag(A);
    diff = tol + 1; //garante a entrada no loop

    //itera até a tolerância ser alcançada
    while diff > tol
        [Q, R] = qr(A); //acha a decomposição qr
        A = R * Q;
        S_new = diag(A);
        diff = norm(S_new - S, 'inf');
        S = S_new;
    end
endfunction
```

A lógica por trás pode ser verificada na página 398 do livro álgebra Linear: Uma Introdução Moderna, de David Poole. Mas, no geral, estamos aplicando uma transformação que aproxima cada vez mais a matriz A de uma triangular superior, até o ponto onde alcançamos a precisão desejada. Bom, agora basta gerar as matrizes para testar o algoritmo. Para isso, usaremos a decomposição espectral, ou seja, $A = Q\Lambda Q^T$, da seguinte forma:

```
eigenvalues1 = [1, 2, 3];
Q1 = orth(rand(3, 3));
D1 = diag(eigenvalues1);
A1 = Q1 * D1 * Q1';
```

```
eigenvalues2 = [4, 5, 6];
Q2 = orth(rand(3, 3));
D2 = diag(eigenvalues2);
A2 = Q2 * D2 * Q2';
```

Aplicando esses comando, as matrizes resultantes foram:

```
 A1  =

   1.4208323  -0.6087083  -0.2077945
  -0.6087083   2.5925951  -0.4928523
  -0.2077945  -0.4928523   1.9865726

--> A2
 A2  =

   5.7148146  -0.1120633  -0.2972575  -0.8223743  -0.8039345
  -0.1120633   6.3200142  -0.6184418  -0.8189118  -1.0436611
  -0.2972575  -0.6184418   6.5327524  -0.601833   -0.4557673
  -0.8223743  -0.8189118  -0.601833    5.4704179   0.6996389
  -0.8039345  -1.0436611  -0.4557673   0.6996389   5.9620009
```

Agora, vamos aplicar a função em cada uma delas e analisar os resultados:

```
--> S = espectro(A1, 10E-6)
 S  =

   2.9999932
   2.0000068
   1.0000000
```

```
--> S = espectro(A2, 10E-6)
 S  =

   8.0000000
   6.9999791
   6.0000209
   5.0000000
   4.0000000
```

Que são exatamente os autovalores das matrizes. Vemos que este é um método muito eficiente para calcular os autovalores, especialmente para matrizes em que outros métodos podem ser muito custosos ou levar a instabilidades numéricas. Para esses casos, uma alternativa eficiente é usar o algoritmo QR para o cálculo dos autovalores.

Gustavo Tironi