Instituto Superior Técnico

Álgebra Linear Numérica

Projeto Computacional

Licenciatura Em Matemática Aplicada e Computação

Junho 2022

Clara Pereira 99405
Marta Sereno 99432
Samuel Pearson 99441

Conteúdo

Grupo I

1.

Neste primeiro exercício é pedido que se implemente o Método de Gram-Schmidt Modificado no *Matlab*. Para tal, foi criada a função *gsm* que recebe uma matriz $A \in \mathbb{R}^{m\times n}$ e determina a respetiva decomposição QR, onde $Q \in \mathbb{R}^{m\times n}$ e $R \in \mathbb{R}^{n\times n}$ é uma matriz triangular superior. Assim, dada a coluna a_i de A, sabe-se que $r_{ii} = \|a_i\|_2$ e que $q_i = \frac{a_i}{r_{ii}}$. Para além disso, dado $i < j \leq n$, pode-se calcular recursivamente os elementos de R acima da diagonal fazendo $r_{ij} = q_i \cdot a_j$ e, de seguida, determina-se $a_{j+1} = a_j - r_{ij} \cdot q_i$ que deverá ser utilizado no próximo cálculo de r_{ij}. Por fim, o programa devolve as matrizes

$$Q = [q_1 \dots q_n] \qquad R = \begin{bmatrix} \mathrm{r}_{11} & r_{12} & \dots & r_{1n} \\ 0 & r_{22} & \dots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & r_{nn} \end{bmatrix}.$$

```
function [Q,R] = gsm(A)

  [m,n] = size(A); % dimensões da matriz A
  Q = zeros(m,n); % matriz genérica com as dimensões de Q
  R = zeros(n,n); % matriz genérica com as dimensões de R
  Anew = A; % gerar matriz de modo a não alterar a matriz A diretamente

  for i = 1:1:n
      R(i,i) = norm(Anew(:,i)); % elementos na diagonal de R
      Q(:,i) = Anew(:,i)/R(i,i); % colunas de Q

      for j = i+1:1:n
          R(i,j) = Q(:,i)'*Anew(:,j); % elementos acima da diagonal de R
          Anew(:,j) = Anew(:,j)-R(i,j)*Q(:,i); % coluna de A a ser usada no
    próximo cálculo de R(i,j), passo de ortogonalização
      end
  end

end
```

No Método de Ortogonalização de Gram-Schmidt Clássico, ortogonaliza-se cada vetor individualmente aos vetores anteriores, enquanto que no Método de Ortogonalização de Gram-Schmidt Modificado, para cada vetor, ortogonaliza-se todos os vetores seguintes ao inicial, como se pode verificar no código acima.

```
>> [Q,R] = gsm([-1 0; 1 -1])

Q =

   -0.7071   -0.7071
    0.7071   -0.7071


R =

    1.4142   -0.7071
         0    0.7071
```

Figura 1: Exemplo de aplicação do Método definido

2. a)

Nesta alínea pretende-se calcular as normas $\|Q_nR_n - A_n\|_F$ e $\left\|Q_n^TQ_n - I_n\right\|_F$, onde $A_n = Q_nR_n$ e $A_n \in \mathbb{R}^{100\times n}$, $n = 10, \ldots, 100$, dada por $(A_n)_{ij} = \frac{1}{i+j-1}$. Foi então criada a função $frob$ que, dado um valor de n, devolve uma lista com os valores de ambas as normas, recorrendo ao comando *norm* com a especificação fro, de Frobenius. Para a primeira norma, a função recorre à decomposição QR determinada através da função *gsm* anterior, enquanto que para a segunda, é definida a matriz I diagonal, com as mesmas dimensões de Q, matriz quadrada $n \times n$.

Função *Matlab* criada para resolução do problema:

```
function [res] = frob(n)

    % definição da matriz A com dimensão 100Xn
    A = zeros(100,n);
    for i = 1:1:100
        for j = 1:1:n
            A(i,j) = 1/(i+j-1);
        end
    end

    % determinação das matriz Q e R através da função definida previamente
    [Q,R] = gsm(A);
    % matriz identidade com as mesmas dimensões de Q
    I = eye(n);

    F1 = (Q*R)-A; % matriz correspondente à primeira norma
    F2 = (Q'*Q)-I; % matriz correspondente à segunda norma

    % determinação de ambas as normas
    res = [norm(F1,"fro"); norm(F2,"fro")];

end
```

Na tabela abaixo apresentam-se os resultados obtidos para cada n:

-	$\|Q_nR_n - A_n\|_F$	$\|Q_n^TQ_n - I_n\|_F$
$n = 10$	0.0000	0.0000
$n = 20$	0.0000	1.6907
$n = 30$	0.0000	2.3720
$n = 40$	0.0000	2.6919
$n = 50$	0.0000	2.8054
$n = 60$	0.0000	2.9485
$n = 70$	0.0000	3.1012
$n = 80$	0.0000	3.2144
$n = 90$	0.0000	3.3588
$n = 100$	0.0000	3.6135

Dado que $A_n = Q_n R_n$ e, sabe-se teoricamente que $Q^T = Q^{-1}$ já que as colunas da matriz Q são ortonormadas, ou seja, $Q^T Q = I$, o resultado deveria ser nulo para ambas as normas. Tal é verdade para a primeira norma. No entanto, como se pode ver nos resultados obtidos para a segunda norma, para matrizes de grandes dimensões, mesmo utilizando o método modificado, esta decomposição funciona mal e não devolve os resultados pretendidos. A acumulação de erro é uma consequência do passo de ortogonalização, onde recorremos à subtração das projeções nas colunas de Q que já foram calculadas. Assim, cada coluna de Q tem uma componente de erro que aumenta com o índice i da coluna, o que explica o aumento de erro com o aumento do valor de n no cálculo de $\left\|Q_n^T Q_n - I_n\right\|_F$, apresentado na tabela acima.

2. b)

Esta alínea tem como objetivo obter a solução exata do sistema $A_{100}x = b$, onde $b \in \mathbb{R}^{100}$ é definido por $b_i = \sum_{j=1}^{100} \frac{1}{i+j-1}$, usando a decomposição QR.
Considerando que

$$Ax = b \Leftrightarrow QRx = b \Leftrightarrow Q^{-1}QRx = Q^{-1}b \Leftrightarrow R^{-1}Rx = R^{-1}Q^{-1}b \Leftrightarrow x = R^{-1}Q^{-1}b,$$

depois de ser definida a matriz A de Hilbert através do comando *hilb* e gerados a sua decomposição QR, coma função *gsm*, e o vetor b, recorrendo ao comando *symsum* que calcula a soma da série pretendida para cada linha i, basta fazer $x = R^{-1}Q^{-1}b$ recorrendo ao comando *inv* que calcula a matriz inversa.

Programa definido para encontrar a solução e tirar conclusões da mesma:

```
% gerar matriz A de hilbert
A = hilb(100);

% gerar vetor b
syms j
b = ones(100,1);
for i = 1:1:100
    b(i) = symsum(1/(i+j-1),j,1,100);
end

% determinar solução do sistema Ax=b usando QR
[Q, R] = gsm(A); % decomposição QR de A
x = inv(R)*inv(Q)*b; % solução deduzida no relatório
disp(x);


% cálculo dos erros relativos da solução
e = ones(100,1);
for k = 1:1:100
    e(k) = abs(1-x(k))*100;
end
disp(e);


% explicar os resultados obtidos
cond(A); % número de condição de A
disp(R); % visualizar matriz R
```

Na tabela abaixo apresentam-se as primeiras 10 linhas da solução obtida e o respetivo erro relativo comparado com a solução prevista $x = [1 \dots 1]^T$,

x	erro relativo
$0.0010 \cdot 10^3$	$0.0000 \cdot 10^5$
$0.0010 \cdot 10^3$	$0.0000 \cdot 10^5$
$0.0010 \cdot 10^3$	$0.0000 \cdot 10^5$
$0.0008 \cdot 10^3$	$0.0002 \cdot 10^5$
$-0.0029 \cdot 10^3$	$0.0039 \cdot 10^5$
$-0.0195 \cdot 10^3$	$0.0205 \cdot 10^5$
$0.0283 \cdot 10^3$	$0.0273 \cdot 10^5$
$-0.1308 \cdot 10^3$	$0.1318 \cdot 10^5$
$-0.0662 \cdot 10^3$	$0.0672 \cdot 10^5$
$-0.4158 \cdot 10^3$	$0.4168 \cdot 10^5$

A estabilidade do Método de Gram-Schmidt Modificado está dependente do número de condição da matriz A, como este número é extremamente elevado, $cond(A) = 4.07 \cdot 10^{19}$, o problema é mal condicionado, o que leva a erros relativos muito altos, como se pode observar nos resultados acima. Recorrendo ao comando *disp*, pode-se, ainda, verificar que a matriz R está muito próxima de ser uma matriz singular, um erro assinalado pelo próprio $Matlab$, já que os seus pivots são muito próximos de zero. Este facto também contribui para o mau condicionamento do problema.

Grupo II

1.

1 a)

Para esta alínea implementou-se a função $doubleSVD$ que recebe como parâmetros uma imagem e um número *opt* (igual a 0 ou 1) e converte a imagem numa matriz A de tipo *double*, devolvendo como resultado a decomposição SVD de A (Matrizes U, S e V) através do comando *svd* do Matlab e, se $opt = 1$, o programa imprime também o número de valores singulares obtidos (tendo em conta que este número corresponde ao mínimo entre o número de linhas e colunas da matriz S, que é diagonal contendo os valores singulares de A).

De seguida apresenta-se o script elaborado e o resultado obtido:

```
function [U,S,V] = doubleSVD(imagem,opt) %opt: opção de fazer o display do
%número de valores singulares obtidos: 1 se queremos display, 0 c.c.
A=im2gray(imread(imagem)); %comando im2gray de modo a considerar a imagem
%como estando a preto e branco

A=double(A);%Obter matriz com entradas com precisão dupla
[U,S,V]=svd(A);%Obter decomposição SVD

if opt==1
    min(size(S)) %mostrar o número de valores singulares obtidos, se desejado
end
end
%Nota: ao aplicar função na command window, para apenas ver o nº de valores
%singulares obtido, usar ";" (suprimindo o output muito grande da
%decomposição SVD da matriz em questão).
```

```
>> doubleSVD('BW_WashingtonDC.jpg',1);

ans =

        1400
```

Figura 2: Nº de valores singulares obtido para a imagem escolhida

1 b)

Para simplificar a imagem e calcular a percentagem de qualidade de dados preservada criou-se a função *simplifica*, que recebe como parâmetros uma imagem e um valor no intervalo $[0, 1]$ (referente à porção de valores singulares preservados) e devolve a imagem simplificada e a sua qualidade, expressa em percentagem.

```
function qualidade = simplifica(imagem,percentagem)

A=im2gray(imread(imagem));
A=double(A);
S=svd(A); %Obter valores singulares de A
Denominador=sumsqr(S);%No cálculo da qualidade da imagem, o denominador é
%a soma dos quadrados dos valores singulares de A
N=ceil(length(S)*percentagem);%min(size(S)) dá-nos o número de valores
% singulares de A. N é o número de valores a preservar.

[U2,S2,V2]=svds(A,N); %Decomposição SVD retendo a percentagem de
% valores singulares desejada
Numerador=sumsqr(S2); %O numerador no cálculo da qualidade é a soma dos
% quadrados dos valores singulares retidos

%Obtenção da nova matriz:
X=U2*S2*V2';
Anew=uint8(X);
%Produção do resultado:
imshow(Anew) %imagem obtida
qualidade=100*(Numerador/Denominador); %qualidade de dados preservada
end
```

Em seguida apresentam-se os resultados obtidos:

Figura 3: Percentagem: 25%

Figura 4: Percentagem: 50%

Figura 5: Percentagem: 75%

Quanto à qualidade das imagens obtidas, os resultados foram os seguintes:

```
>> simplifica('BW_WashingtonDC.jpg',0.25)

ans =

   99.7198

>> simplifica('BW_WashingtonDC.jpg',0.5)

ans =

   99.9887

>> simplifica('BW_WashingtonDC.jpg',0.75)

ans =

   99.9995
```

Figura 6: Qualidade

Podemos então verificar, tanto pelas percentagens obtidas como pela visualização das imagens simplificadas, que a qualidade das imagens se manteve bastante elevada, mesmo preservando apenas 25% dos valores singulares, o que indica que a eliminação dos valores singulares de menor valor absoluto pouco contribuiu para a perda de definição da imagem.

2.

De modo a simplificar agora imagens a cores, implementou-se a função *cores*, que recebe como inputs uma imagem (a cores) e um parâmetro *percentagem* (que na verdade deverá ser um número real no intervalo $[0, 1]$ e não uma percentagem propriamente dita), devolvendo a imagem resultante de preservar apenas essa mesma percentagem dos valores singulares em cada camada *RGB*.

O algoritmo encontra-se devidamente comentado no código:

```
function res = cores(imagem,percentagem) %Nota: na verdade deve-se usar
%como input a fração dos valores singulares que se pretende preservar
%(número entre 0 e 1) e não uma percentagem propriamente dita.

%carregar imagem:
A=imread(imagem);

%decompor em camadas de diferentes cores e obter as matrizes com entradas
% em precisão dupla:
A1=A(:,:,1); %Vermelho
A1=double(A1);
A2=A(:,:,2); %Verde
A2=double(A2);
A3=A(:,:,3); %Azul
A3=double(A3);

%Obter valores singulares de cada matriz:
s1=svd(A1);
s2=svd(A2);
s3=svd(A3);
%Obter o número de valores singulares a preservar em cada matriz
%(menor valor inteiro que permite preservar pelo menos a percentagem
%pretendida):
n1=ceil(length(s1)*percentagem);
n2=ceil(length(s2)*percentagem);
n3=ceil(length(s3)*percentagem);

%Obter a decomposição SVD de cada matriz com o número certo de valores
%singulares preservados:
[U1,S1,V1]=svds(A1,n1);
[U2,S2,V2]=svds(A2,n2);
[U3,S3,V3]=svds(A3,n3);

%Obter as novas camadas RGB:
X1=U1*S1*V1';
X2=U2*S2*V2';
X3=U3*S3*V3';

Anew1=uint8(X1);
Anew2=uint8(X2);
Anew3=uint8(X3);

%Visualizar a imagem resultante da compressão:
Aend(:,:,1)=Anew1;
Aend(:,:,2)=Anew2;
Aend(:,:,3)=Anew3;
imshow(Aend)
```

Resultados (input - cores('RGB_Passaro.JPG',percentagem):

Figura 7: Percentagem: 25%

Figura 8: Percentagem: 50%

Figura 9: Percentagem: 75%

Novamente, observa-se que, em termos visuais, as imagens parecem permanecer idênticas quando se preservam 25%, 50% ou 75% dos valores singulares em cada camada *RGB*, indicando que a qualidade destas deverá ser fortemente ditada por uma pequena percentagem dos valores singulares mais elevados. Apenas a partir de 10-15% de retenção é que se torna evidente a perda de definição da imagem.

Grupo III

1.

A primeira parte deste exercício consistia em implementar, em *Matlab*, uma função que aproximasse o número de condição de uma matriz $A \in \mathbb{R}^{n \times n}$, simétrica e não singular. Sabemos que, nesse caso,

$$cond_2(A) = \rho(A)\rho(A^{-1}),$$

em que $\rho(A)$ representa o raio espetral da matriz A, isto é, o módulo do máximo valor próprio desta matriz, em módulo. Para efetuarmos o cálculo de $\rho(A)$, recorremos ao método iterativo das potências, que, dada uma aproximação x_0 tal que $||x_0||_\infty = 1$, uma matriz A, e uma tolerância ϵ, devolve uma aproximação de $\rho(A)$. A função *mpot*2 recebe então esses 3 argumentos, e aplica o método das potências tanto para A como para A^{-1}, multiplicando, finalmente, os valores obtidos.

```
function [c] = mpot2(A,x0,eps)
if norm(x0,Inf)~= 1 %verificar condições para x0: norma infinito = 1
    disp("a aproximação inicial escolhida não é válida")
else
    c=1; %inicializar número de condição
    for i= 1:2
        if i==1
            B=A; %calcular rho(A)
        elseif i==2
            B=inv(A); %calcular rho(A^-1)
        end
        x=x0; %inicializar método das potências: k=0
        d=eps+1;
        lb1=eps;
        while d>eps %iterações do método: k=1,2,...
        lb0=lb1;
        z=B*x;
        v=find(x); %lista de índices de x(k) diferentes de 0
        j=v(1);
        lb1=z(j)/x(j);
        x=z/lb1;
        d=abs((lb1-lb0)/lb1);
        end
        c=abs(lb1)*c; %no final do ciclo for, c=1*rho(A)*rho(A^-1)
    end
end
```

```
w =

   1.0e+18 *

  Columns 1 through 9

    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0005    0.0163    0.3618

  Columns 10 through 16

    0.6981    0.4374    0.9901    6.2696    1.7347    3.0226    3.1975
```

Figura 10: números de condição

2.

Para a segunda parte do Exercício 3, testou-se o código anterior no cálculo do número de condição de matrizes de Hilbert com diferentes dimensões. Pretende-se verificar que o número de condição aumenta exponencialmente quando n aumenta. Para isto, calculou-se o número de condição das matrizes de Hilbert com $n = 5, ..., 15, 20, 25, 30, 35, 40$, pela função *metpot2*.

```
function [c] = mpot2(A,x0,eps)
if norm(x0,Inf)~= 1 %verificar condições para x0: norma infinito = 1
    disp("a aproximação inicial escolhida não é válida")
else
    c=1; %inicializar número de condição
    for i= 1:2
        if i==1
            B=A; %calcular rho(A)
        elseif i==2
            B=inv(A); %calcular rho(A^-1)
        end
        x=x0; %inicializar método das potências: k=0
        d=eps+1;
        lb1=eps;
        while d>eps %iterações do método: k=1,2,...
        lb0=lb1;
        z=B*x;
        v=find(x); %lista de índices de x(k) diferentes de 0
        j=v(1);
        lb1=z(j)/x(j);
        x=z/lb1;
        d=abs((lb1-lb0)/lb1);
        end
        c=abs(lb1)*c; %no final do ciclo for, c=1*rho(A)*rho(A^-1)
    end
end
```

De seguida, traçou-se o gráfico dos logaritmos dos respetivos números de condição obtidos, em função da dimensão n das matrizes. Repetiu-se o processo com a função *cond* do *Matlab*, de forma a verificar a precisão dos resultados. Espera-se obter um gráfico linear ou aproximadamente linear.

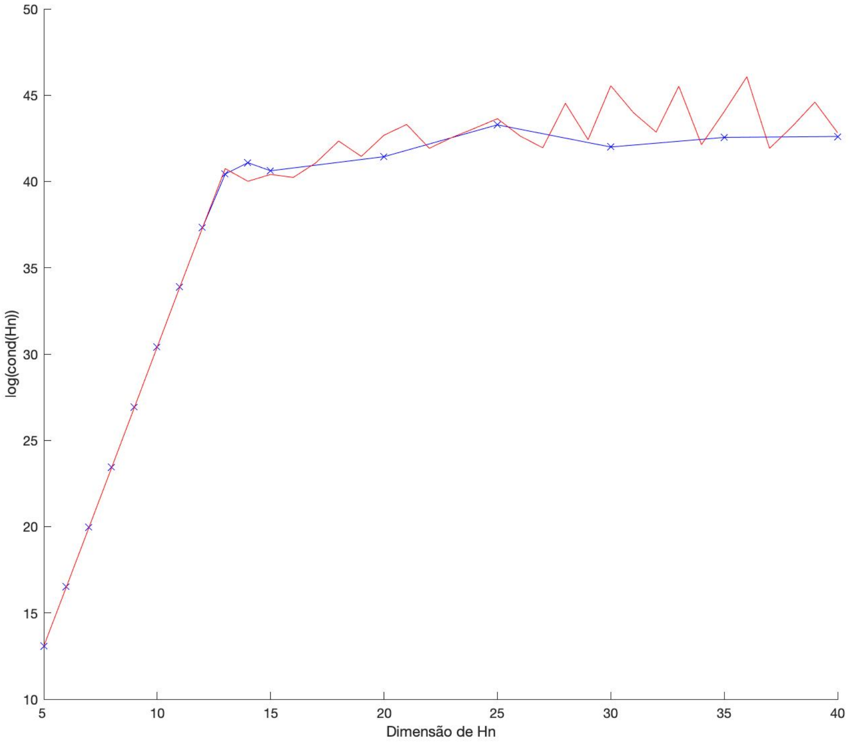

Figura 11: *mpot*2, a azul, e *cond*, a vermelho

Analisando os resultados, podemos concluir que, até $n = 13$, o número de condição tem crescimento exponencial (o seu logaritmo tem crescimento linear), todavia parece tornar-se relativamente constante para n superior. As matrizes de Hilbert têm, no entanto, um crescimento exponencial, mesmo para $n > 13$, que pode ser demonstrado através de processos analíticos. Assim, o método não devolve resultados precisos para matrizes de Hilbert de dimensões superiores a 13. Este fenómeno deve-se à elevada propagação de erros, que ocorre ao calcular a inversa da matriz de Hilbert, que é quase singular. Efetivamente, as matrizes de Hilbert apresentam um problema pelo seu mau condicionamento, sendo estas tão sensíveis que a função implementada, bem como a função *cond* do $Matlab$, devolvem resultados drasticamente distintos quando avaliadas em computadores diferentes.

Bibliografia

[1] Alfio Quateroni, Riccardo Sacco, Fausto Saleri (2007), *Numerical Mathematics*

[2] N. S. Hoang and A. G. Ramm (2007), *Solving ill-conditioned linear algebraic systems by the dynamical systems method,* Department of Mathematics, Kansas State University

[3] Cleve Moler, (2013), *Hilbert Matrices,* Mathworks

[4] Brady Mathews, (2014), *Image Compression Using Singular Value Decomposition (SVD)*, The University of Utah