P3Q1 MAP22010

Gustavo Soares e Pedro Akira Kitayama

Julho de 2022

Conteúdo

1. Introdução

Este relatório tem por objetivo descrever a implementação dos algoritmos da fatoração QR por Gram-Schmidt clássico, Gram-Schimidt modificado e por Householder. Iremos calcular o tempo de execução e a memória RAM utilizada, além de comparar os erros dos algoritmos no que diz respeito à qualidade da fatoração e da ortonormalização.

2. Relembrando a teoria: Fatoração QR

Lembremos da teoria da fatoração. Segundo [1], toda matriz A mxn com posto máximo pode ser fatorada na forma

$$A = QR.$$

Onde Q é uma matriz com vetores coluna **ortonormais** e R é uma matriz nxn triangular superior invertível. Automaticamente, tiramos que Q satisfaz:

$$Q^T Q = I$$

Assim como comenta [2], a fatoração QR é muito semelhante à fatoração LU já vista no curso, com a única diferença que Q possui colunas ortonormais. Ainda em [2], encontramos um bom exemplo para ilustrar a forma dessa fatoração:

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \\ 1/\sqrt{2} & -1/\sqrt{2} & 0 \end{bmatrix} \cdot \begin{bmatrix} \sqrt{2} & 1/\sqrt{2} & \sqrt{2} \\ 0 & 1/\sqrt{2} & \sqrt{2} \\ 0 & 0 & 1 \end{bmatrix} = QR$$

3. Implementando os Algoritmos

Existem diversos tipos de implementação prática dessa fatoração, em particular trabalharemos com os algoritmos de Gram Schmidt (clássico e modificado) e de Householder. O enfoque deste relatório está na implementação destes algoritmos e não em suas deduções, o leitor mais assíduo poderá encontrar a teoria por trás dos algoritmos citados em [3],[4],[5], respectivamente.

QR via Gram-Schimidt Clássico

Como referência, estaremos tomando o artigo sobre fatorações de matrizes no blog do Matemático Cleve Moler, que pode ser acessado em [6]. Lá é apresentado o algoritmo da fatoração via Gram-Schmidt clássico feito na linguagem Matlab, segue abaixo a tradução do algoritmo para a linguagem Python.

```
1
2  #IMPLEMENTA QR POR GRAM-SCHMIDT CLASSICO
3
4  def gs(A):
5
6    #INICIA CONTAGEM DE TEMPO
7    inicio = time.time()
8
9    #CAPTA O NUMERO DE LINHAS E DE COLUNAS DE A
10   num_linhas = np.shape(A)[0]
11   num_colunas = np.shape(A)[1]
12
13   #CRIA Q,R
```

```
14    Q = np.zeros(num_linhas,num_colunas,dtype = 'float')
15    R = np.zeros(num_linhas_num_colunas,dtype = 'float')
16
17    #PERCORRE AS COLUNAS
18    for k in range(num_colunas):
19
20      #IGUALA AS COLUNAS DE Q AS DE A
21      Q[:,k] = A[:,k]
22
23      #ORTOGONALIZA VIA GRAM SMITH
24      if k != 0:
25        R[0:k,k] = np.matmul(np.transpose(Q[:,k-1]),Q[:,k])
26        Q[:,k] = Q[:,k] - np.matmul(Q[:,0:k],R[0:k,k])
27
28
29      R[k,k] = np.linalg.norm(Q[:,k])
30      Q[:,k] = Q[:,k]/R[k,k]
31
32    #FINALIZA TEMPO
33    fim = time.time()
34    return Q,R,(fim-inicio)
```

A função retorna as matrizes Q e R encontradas, além do tempo de execução da função em segundos. Para testar a função, geramos uma matriz aleatória A com a biblioteca numpy, obtivemos o produto QR e avaliamos três versões de normas para a matriz $D = A - QR$, cujas definições podem ser encontradas em [7],[8],[9]. Os resultados obtidos seguem na tabela abaixo.

```
1  ------------------------------------------------------------
2  NORMA DO MAXIMO DE D = A-QR = 1.2628786905111156e-15
3  ------------------------------------------------------------
4  NORMA DE FROBENIUS DE D = A-QR = 1.3577965726176163e-15
5  ------------------------------------------------------------
6  NORMA 1 DE D = A-QR = 1.970645868709653e-15
7  ------------------------------------------------------------
```

Podemos concluir, então, que a implementação desse primeiro algoritmo foi feita de forma satisfatória. Partamos para a próxima implementação.

QR via Gram-Schmidt Modificado

Esta fatoração também é apresentada em [6] e sua implementação tanto em Python quanto em Matlab é muito semelhante à do algoritmo do método clássico, como podemos ver abaixo.

```
#IMPLEMENTA QR POR GRAM-SCHMIDT MODIFICADO

def modified_gram_smith(A):

  #INICIA A CONTAGEM DE TEMPO
  inicio = time.time()

  #CAPTA O NMERO DE LINHAS E COLUNAS DE A
  num_linhas = np.shape(A)[0]
  num_colunas = np.shape(A)[1]

  #CRIA MATRIZES Q,R
  Q = np.array(num_linhas,num_colunas,dtype = 'float')
  R = np.array(num_linhas,num_colunas,dtype='float')

  #PERCORRE COLUNAS
  for k in range(num_colunas):

    #IGUALA COLUNA DE Q COM A DE A
    Q[:,k] = A[:,k]

    for i in range(k):

      R[i,k] = np.matmul(np.transpose(Q[:,i]),Q[:,k])
      Q[:,k] = Q[:,k] - R[i,k]*Q[:,i]

    R[k,k] = np.linalg.norm(Q[:,k])
```

```
28      Q[:,k] = Q[:,k]/R[k,k]
29
30    #FINALIZA TEMPO
31    fim = time.time()
32    return Q,R,(fim-inicio)
```

Analogamente, a função retorna as matrizes Q e R e o tempo de execução da função em segundos. O teste foi feito da mesma maneira: geramos uma matriz aleatória A de dimensão 10x10 e calculamos via função as matrizes Q e R. Comparamos então sob as diferentes normas a matriz de diferença $A - QR$.

```
1 ------------------------------------------------------------
2 NORMA DO MAXIMO DE D = A-QR = 6.869504964868156e-16
3 ------------------------------------------------------------
4 NORMA DE FROBENIUS DE D = A-QR = 6.698571303802826e-16
5 ------------------------------------------------------------
6 NORMA 1 DE D = A-QR = 9.853229343548264e-16
7 ------------------------------------------------------------
```

Como pode ser visto, a implementação mostra-se satisfatória.

QR via HouseHolder

Aqui estaremos utilizando uma implementação alternativa que encontramos no Github, ela pode ser acessada em [10]. Embora haja algumas diferenças, na prática o procedimento usado é o mesmo do que o algoritmo de fatoração QR de HouseHolder mostrada no blog. O código equivalente para a linguagem Python do algoritmo segue abaixo.

```
1 #IMPLEMENTA QR POR HOUSEHOLDER
2
3 # CONVERTE UM ARRAY 1D EM UM VETOR COLUNA
4
5 def column_convertor(x):
6
```

```
7       x.shape = (1, x.shape[0])
8       return x
9
10  # RETORNA A NORMA
11
12  def get_norm(x):
13
14      return np.sqrt(np.sum(np.square(x)))
15
16  # RETORNA A MATRIZ DE HOUSEHOLDER DO VETOR X DADO
17
18  def householder_transformation(v):
19
20      size_of_v = v.shape[1]
21      e1 = np.zeros_like(v)
22      e1[0, 0] = 1
23      vector = get_norm(v) * e1
24      if v[0, 0] < 0:
25          vector = - vector
26      u = (v + vector).astype(np.float64)
27      H = np.identity(size_of_v) -
28      ((2*np.matmul(np.transpose(u), u))/np.matmul(u, np.transpose(u)))
29
30      return H
```

Na parte a seguir, calcula-se a fatoração em si. A função retorna diretamente as matrizes Q e R, além do tempo de execução em segundos.

```
1   # RETORNA MATRIZES Q,R
2   def qr_step_factorization(q, r, iter, n):
3
4       v = column_convertor(r[iter:, iter])
5       Hbar = householder_transformation(v)
6       H = np.identity(n)
7       H[iter:, iter:] = Hbar
8       r = np.matmul(H, r)
```

```
9       q = np.matmul(q, H)
10
11      return q, r
12
13  def fatHouseholder(A):
14
15      #INICIA TEMPO
16      inicio = time.time()
17
18      #CAPTA AS DIMENSOES DE A
19      n = np.shape(A)[0]
20      m = np.shape(A)[1]
21
22      #INICIALIZA Q,R
23      Q = np.identity(n)
24      R = A.astype(np.float64)
25
26      for i in range(min(n, m)):
27          #PARA CADA ITERACAO, MATRIZ H E CALCULADA PRA LINHA i+1
28          Q, R = qr_step_factorization(Q, R, i, n)
29
30      min_dim = min(m, n)
31
32
33      R = R[:min_dim, :min_dim]
34
35
36      #FINALIZA TEMPO
37      fim = time.time()
38
39      return Q,R,(fim-inicio)
```

O teste foi feito da mesma maneira: geramos uma matriz aleatória A de dimensão 10x10 e calculamos via função as matrizes Q e R. Comparamos então sob as diferentes normas a matriz de diferença $A - QR$.

```
NORMA DO MAXIMO DE D = A-QR = 1.326716514427062e-14
-------------------------------------------------
NORMA DE FROBENIUS DE D = A-QR = 6.317458022461165e-15
-------------------------------------------------
NORMA 1 DE D = A-QR = 7.355227538141662e-15
-------------------------------------------------
```

Como é possível notar, os erros são irrisórios, então pode-se considerar a implementação do algoritmo satisfatória.

4. Matrizes Cobaias

Para testar a qualidade e a eficiência das soluções dos algoritmos anteriores, estaremos utilizando dois tipos de matrizes.

Matrizes Mágicas

As matrizes mágicas remontam desde a China antiga e possuem diversos significados astrológicos interessantes que podem ser encontrados em [11]. Matematicamente, uma matriz mágica é tal que a soma de cada coluna, de cada linha e das suas diagonais são iguais, como no exemplo abaixo.

$$M = \begin{bmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{bmatrix}$$

Para a geração de matrizes mágicas em nosso programa, estaremos utilizando a biblioteca magic_square, que se mostrou bastante precisa e rápida em nossos testes.

Matrizes de Hilbert

Matrizes de Hilbert foram introduzidas pelo famoso matemático David Hilbert e são notórias por terem mau condicionamento, por isso são matrizes 'dificeis' de lidar em computação numérica. O leitor interessado pode consultar [12] para encontrar a definição de condicionamento e o seu estudo para as matrizes de Hilbert em particular. Tais matrizes são definidas por $H_{ij} = \frac{1}{i+j-1.}$. Segue abaixo um exemplo de uma matriz

de Hilbert de ordem 5.

$$H = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{bmatrix}$$

A implementação de uma função que gera esse tipo de matriz é simples e foi feita para ser usada em nosso programa, como mostra o código abaixo.

```
1  #FUNCAO PARA DEVOLVER MATRIZ DE HILBERT
2
3  def hilbert_matrix(order):
4
5    h_matrix = np.zeros((order,order))
6
7    for i in range(order):
8      for j in range(order):
9
10       h_matrix[i][j] = 1/((i+1)+(j+1)-1)
11
12   return h_matrix
```

5. Condicionamento das Matrizes Cobaias

De acordo com [13], o número de condição de uma **matriz** A (não do problema!) sob a norma $|| \cdot ||$ é definido por

$$k(A) = ||A|| \cdot ||A^{-1}||.$$

Quando $k(A)$ é pequeno dizemos que a matriz A é bem condicionada, enquanto que se $k(A)$ é alto dizemos que a matriz A é mal condicionada. Em particular, em nosso programa estaremos utilizando a norma do máximo para medir o condicionamento de nossas matrizes a partir da função $linalg.cond()$ da biblioteca numpy.

Matrizes de Hilbert

Como já citado, as matrizes de Hilbert são famosas por serem mal condicionadas, e em nosso programa isso se mostrou evidente, pois a média de condicionamento de todas as matrizes de Hilbert geradas em nosso programa foi de $1.9141681576621396e + 22$

Matrizes Mágicas

As matrizes mágicas de ordem ímpar possuem a propriedade de serem bem condicionadas, enquanto que as matrizes mágicas de ordem par geralmente são mal condicionadas, de acordo com os nossos testes. Dessa maneira, optamos por testar apenas matrizes mágicas de ordem par para montar a tabela de matrizes mal-condicionadas em nosso programa. A média de condicionamento dessas matrizes mágicas foi de inf.

Já sobre a tabela de matrizes bem condicionadas, utilizamos matrizes mágicas de ordem ímpar e a média do número de condicionamento foi de 177.9920275366374.

6. Testando os Algoritmos

Aqui iremos testar a eficiência de cada algoritmo visto. Estaremos utilizando a norma do máximo para nos situarmos no pior caso possível do erro.

Na teoria, sabemos que $A = QR$ e $Q^TQ = I$, portanto estaremos avaliando as normas $||A - QR||$ e $||I - Q^TQ||$, além do tempo de execução e a quantidade de memória RAM gasta. A seguir faremos uma análise dos resultados obtidos. Dessa forma, o leitor interessado pode conferir os as tabelas com os resultados na seção resultados.

Qualidade da fatoração

Como pode-se notar no gráfico abaixo, o erro da fatoração por Gram-Schimidt clássico aumenta com o tamanho de n (dimensão da matriz quadrada de teste). Já o erro da fatoração de Householder aumenta de uma forma mais suave (e insconstante) com o tamanho de n e a fatoração por Gram-Schimidt modificado tem um erro que é praticamente irrisório comparado com as outras duas fatorações. Assim, no quesito qualidade da fatoração, a fatoração QR por Gram-Schimidt modificado leva vantagem sobre as outras.

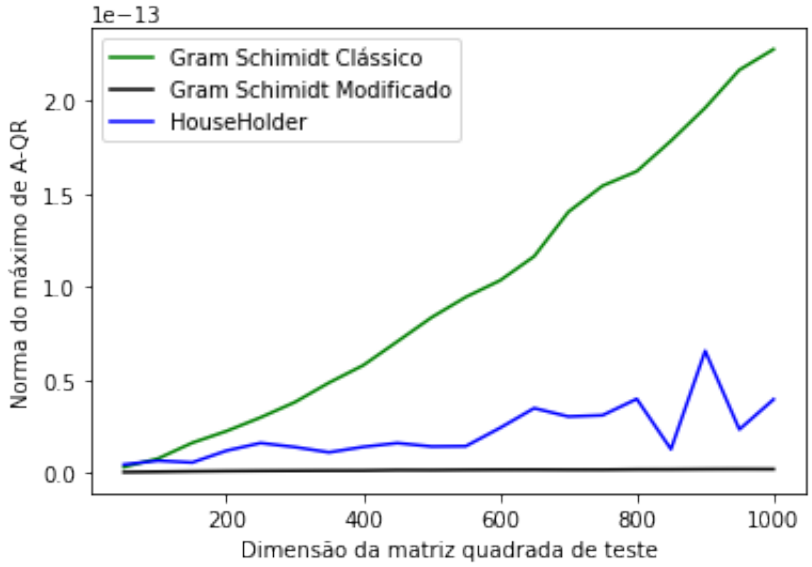

Figura 6.1: Gráfico da norma dos erros A-QR pela dim. da matriz

Qualidade das ortonormalizações

Como podemos observar no gráfico abaixo, a fatoração por Gram - Schimdt clássico apresenta um erro muito alto com o crescimento de n, já o processo por Gram-Schimidt modificado apresenta taxa de erro menor, porém ainda maior que a norma do erro da fatoração via Householder, portanto essa última fatoração leva vantagem no quesito de preservar a ornormalização da matriz Q.

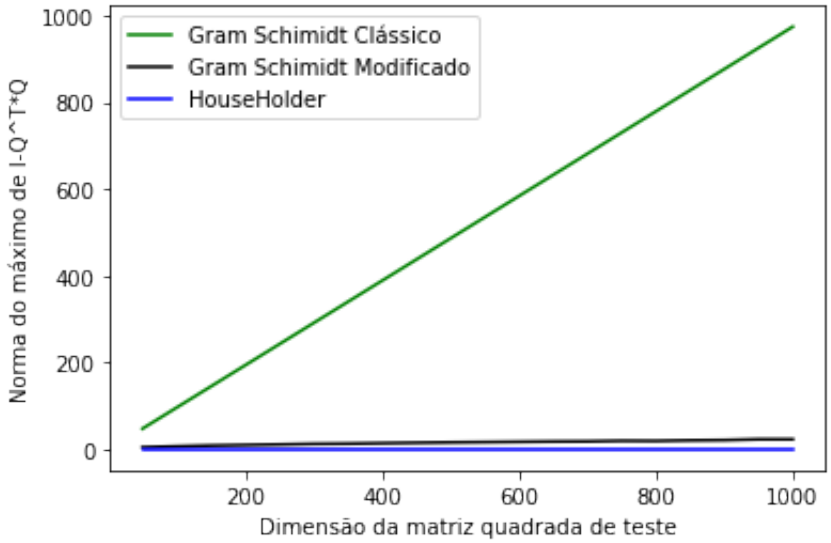

Figura 6.2: Gráfico da norma dos erros $I - Q^T Q$ pela dim. da matriz

Tempo de execução

Observamos na figura 6.3 que a fatoração de Householder mostra-se muito mais demorada para matrizes de dimensões maiores que 400 em relação às fatorações de Gram-Schimidt, pois há um crescimento exponencial com o tamanho da dimensão n da matriz. Neste último grupo, a fatoração de Gram-Schimidt clássico leva um menor tempo do que a fatoração pelo algoritmo modificado, mas cresce de forma mais linear do que a de Householder.

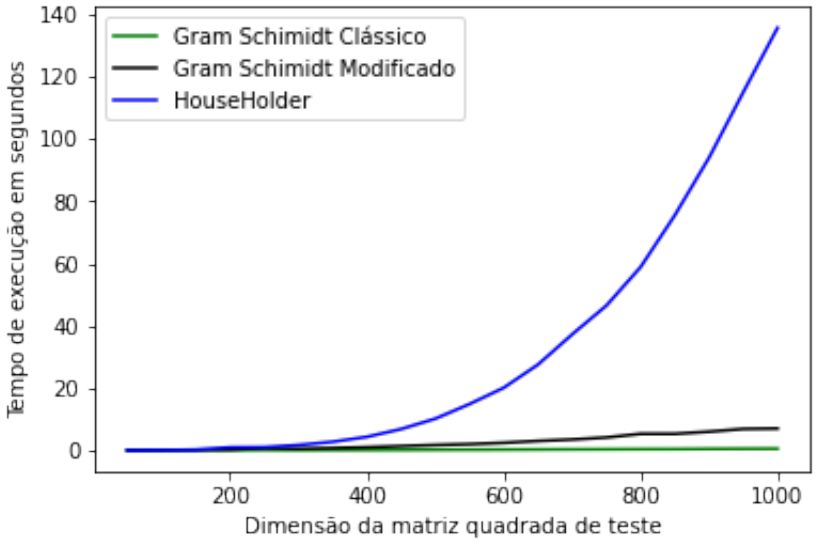

Figura 6.3: Gráfico do tempo de execução pela dim. da matriz

Para matrizes quadradas com dimensões menores que 400, podemos observar que a diferença de tempo entre os três algoritmos não se mostra tão discrepante, ficando abaixo de 20s.

Memória RAM gasta

Os diagnósticos da memória RAM gasta pelas funções tomando como base uma matriz aleatória $1000x1000$ estão em arquivos .txt no rar compactado no e-disciplinas. Podemos observar que gastou-se a maioria da memória RAM criando-se as estruturas do numpy (principalmente arrays), enquanto que o tempo d execução se concentrou na parte dos algoritmos em si.

7. Resultados

Para armazenar os resultados obtidos, criamos uma tabela que contém informações sobre a dimensão das matrizes usadas nos testes, seu condicionamento, as normas das diferenças citadas, além do tempo de execução do algoritmo para aquela matriz.

Devido o tamanho das tabelas, elas poderão ser acessadas em arquivo .txt no arquivo compactado enviado via e-disciplinas: o primeiro arquivo diz respeito às matrizes mal condicionadas: as de Hilbert e as mágicas de ordem par, enquanto que o segundo diz respeito às matrizes bem condicionadas: mágicas de ordem ímpar.

8. Aplicações

8.1 Determinação de poços em redes de aquíferos freáticos [14]

Em geologia, aquíferos são formações que contém água e permitem que quantidades significativas dessa água se movimentem em seu interior em condições naturais. Para determinar os poços principais que podem ser usados, é possível usar a fatoração QR com dados de variabilidade piezométrica (altura que medeia da superfície de um terreno a um lençol de água) de redes de monitoramento.

8.2 MIMO [15]

MIMO, ou multiple-input and multiple-output se trata do uso de múltiplas antenas de rádio tanto no transmissor quanto no receptor para melhorar a performance da comunicação. Nesse método de tecnologia de antenas, que também é utilizado para outros métodos de transmissão como wifi e 4g, um transmissor envia múltiplos fluxos de sinais através de múltiplas antenas, e esses sinais passam por uma matriz de todos os caminhos que podem percorrer entre antenas transmissoras e receptoras a qual é decodificada pelos receptores através da fatoração QR.

8.3 Mínimos Quadrados [16]

Para resolver uma equação $X\beta = Y$ em β com mais observações (Y) que variáveis (X), o método de mínimos quadrados é utilizado de modo a encontrar $\hat{\beta}$ com menor erro através de $\sum(Y - \hat{Y})^2 = \sum(Y - X\hat{\beta})^2$. É possível usar fatoração QR para fazer $X = QR$ onde $Q^T = Q^{-1}$ e R é invertível e então pode-se inverter X de modo a resolver a equação na forma

$$\hat{\beta} = (QR)^{-1}Y = R^{-1}Q^TY$$

Bibliografia

[1] Shores, Thomas. Applied Linear Algebra and Matrix Analysis.1.ed.2007.p.338.

[2] Strang, Gilbert. Algebra Linear e suas aplicações.Tradução da 4 ed.2018.p.181.

[3] Shores, Thomas. Applied Linear Algebra and Matrix Analysis.1.ed.2007.p.323.

[4] Trefethen,Lloyd;Bau,David.Numerical Linear Algebra.1.ed.Lecture 8.p.57.

[5] Trefethen,Lloyd;Bau,David.Numerical Linear Algebra.1.ed.Lecture 8.p.69.

[6] https://blogs.mathworks.com/cleve/2016/07/25/compare-gram-schmidt-and-householder-orthogonalization-algorithms/

[7] Shores, Thomas. Applied Linear Algebra and Matrix Analysis.1.ed.2007.p.306.

[8] Shores, Thomas. Applied Linear Algebra and Matrix Analysis.1.ed.2007.p.310.

[9] Shores, Thomas. Applied Linear Algebra and Matrix Analysis.1.ed.2007.p.306.

[10] https://gist.github.com/Hsankesara/cd35edb30825df19f182a6ecf96e126e

[11] Schinz, Alfred. The Magic Square: Cities in Ancient China. Edition Axel Menges.1996.p. 428.

[12] https://proceedings.sbmac.org.br/sbmac/article/download/2640/2659

[13] Trefethen,Lloyd;Bau,David.Numerical Linear Algebra.1.ed.Lecture 12.p.94.

[14] https://repositorio.unb.br/bitstream/10482/9021/1/2008_SusanneTainaRamalhoMaciel.pdf

[15] https://people.duke.edu/ hpgavin/SystemID/References/Tam-QR-history-2010.pdf

[16] https://towardsdatascience.com/qr-matrix-factorization-15bae43a6b2