UNIVERSIDADE FEDERAL DO AMAZONAS

PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA ELÉTRICA

IMPLEMENTAÇÃO DO ALGORITMO DO PROCESSO DE GRAN-SCHMIDT

Andevaldo da Encarnação Vitório

MANAUS-AM

2024

Andevaldo da Encarnação Vitório

IMPLEMENTAÇÃO DO ALGORITMO DO PROCESSO DE GRAN-SCHMIDT

Este trabalho foi preparado como parte dos requisitos da disciplina *Sistemas Lineares* oferecida pelo Programa de Pós-graduação em Engenharia Elétrica da Universidade Federal do Amazonas.

Prof. Dr. João Edgar Chaves Filho

MANAUS-AM

2024

Capítulo 1

Processo de Gran-Schmidt

Este capítulo apresenta a descrição do código que realiza uma das operações essenciais de álgebra linear: a ortogonalização de vetores e a obtenção da fatoração QR de uma matriz. Para isso, foi utilizada a linguagem *Python* e a sua biblioteca *NumPy*. A função principal é a `gram_schmidt_modified`, que ortogonaliza um conjunto de vetores e pode ser utilizada para calcular a matriz Q ortonormal e a matriz triangular superior R da fatoração QR.

1.1 Algoritmo de Gram-Schmidt Clássico

O algoritmo de Gram-Schmidt é utilizado para ortogonalizar um conjunto de vetores, convertendo-os em uma base ortogonal. Para isso, ele utiliza um processo de projeção, no qual cada vetor do conjunto é projetado sobre os vetores já ortogonais para garantir que os novos vetores sejam ortogonais entre si. A seguir, apresentamos o algoritmo clássico de Gram-Schmidt, seguido de sua implementação em Python.

1.1.1 Definições Matemáticas

O algoritmo de Gram-Schmidt transforma um conjunto de vetores linearmente independentes em um conjunto ortogonal. Dado um conjunto de vetores $\{v_1, v_2, \ldots, v_n\}$, o processo de ortogonalização gera um conjunto ortogonal $\{u_1, u_2, \ldots, u_n\}$, onde:

$$u_1 = v_1 \tag{1.1}$$

E, para $i > 1$:

$$u_i = v_i - \text{proj}_{u_1}(v_i) - \text{proj}_{u_2}(v_i) - \cdots - \text{proj}_{u_{i-1}}(v_i) \tag{1.2}$$

onde a projeção de um vetor v sobre um vetor u é dada por:

$$\text{proj}_u(v) = \frac{v \cdot u}{u \cdot u} u \tag{1.3}$$

Após a ortogonalização, os vetores podem ser normalizados para garantir que tenham norma unitária, transformando-os em uma base ortonormal.

1.2 Implementação em Python

Baseado nas definições do algoritmo, foi realizada a sua implementação em *Python*. A seguir, apresentamos o código que executa a ortogonalização e a normalização de um conjunto de vetores.

```
import numpy as np


def proj(v, u):
    """
    Retorna o projeto do vetor v sobre o vetor u.
    """
    return (v @ u) / (u @ u) * u


def gram_schmidt(vectors):
    """
    Algoritmo de Gram-Schmidt para ortogonalizar um conjunto de vetores.

    Parameters:
        vectors (np.ndarray): Conjunto de vetores a serem ortogonalizados.

    Returns:
        np.ndarray: Conjunto de vetores ortogonais.
    """
    u = np.copy(vectors)  # Cria uma cópia dos vetores
    for i in range(len(vectors)):
        for j in range(i):
            u[i] -= proj(vectors[i], u[j])  # Subtrai a projeção
    return u


def normalize(vectors):
    """
```

```
  Normaliza um conjunto de vetores. Caso o vetor seja nulo, ele é retornado sem alterações.

  Parameters:
      vectors (np.ndarray): Conjunto de vetores a serem normalizados.

  Returns:
      np.ndarray: Vetores normalizados.
  """
  norms = np.linalg.norm(
      vectors, axis=1)  # Calcula as normas de todos os vetores
  # Normaliza os vetores, substituindo os nulos por si mesmos
  return np.divide(vectors.T, norms, where=norms != 0).T


# Exemplo de uso
vectors = np.array([[12., -51., 4.],
                    [6., 167., -68.],
                    [-4., 24., -41]])

ortho_basis = gram_schmidt(vectors)
ortonormal_basis = normalize(ortho_basis)

# Verificando ortonormalidade
print("Base Ortonormal:")
for i, u in enumerate(ortonormal_basis):
  print(f'u[{i}] = ', np.round(u, 2))

# Verifica a ortonormalidade, o produto interno deve ser a matriz identidade
print('\n', np.round(ortonormal_basis.T @ ortonormal_basis, 2))
```

Código 1.1: Implementação do algoritmo de Gram-Schmidt para ortogonalização e normalização de vetores.

1.2.1 Descrição do Funcionamento do Código

O código implementa o algoritmo de Gram-Schmidt para ortogonalizar e normalizar um conjunto de vetores. A função `proj` calcula o projeto de um vetor v sobre um vetor u, utilizando a fórmula matemática de projeção. A função `gram_schmidt` realiza a ortogonalização do conjunto de vetores de entrada. Para cada vetor v_i, o código subtrai as projeções de todos os vetores anteriores $\{u_1, u_2, \ldots, u_{i-1}\}$, garantindo que os vetores resultantes sejam ortogonais entre si. A função `normalize` normaliza os vetores ortogonais, ou seja, divide cada vetor pela sua norma para garantir que todos os vetores tenham módulo unitário. Após a execução do algoritmo, a base ortonormal é exibida e verificada. O produto interno entre os vetores da base ortonormal deve resultar na matriz identidade, confirmando que os vetores são, de fato, ortogonais e normalizados.

1.2.2 Saída Esperada

A saída do código será a exibição dos vetores ortonormais com duas casas decimais, bem como a matriz resultante do produto interno entre os vetores da base ortonormal. O produto interno deve ser uma matriz identidade, indicando que os vetores são ortogonais e normalizados corretamente. A seguir está apresentada a saída do código apresentado:

```
Base Ortonormal:
u[0] =  [ 0.23 -0.97  0.08]
u[1] =  [ 0.62  0.08 -0.78]
u[2] =  [-0.75 -0.23 -0.62]

 [[ 1.  0.  0.]
 [ 0.  1. -0.]
 [ 0. -0.  1.]]
```

A matriz identidade confirmará a ortonormalidade da base obtida.

1.3 Algoritmo de Gram-Schmidt Modificado

O algoritmo de Gram-Schmidt modificado é uma versão otimizada do algoritmo clássico, onde os vetores ortogonais são calculados de maneira incremental. Em vez de recalcular as projeções a cada iteração, ele ajusta o vetor v_i diretamente com a subtração das projeções sobre os vetores ortogonais já calculados, sem a necessidade de armazenar as projeções.

1.3.1 Definições Matemáticas

Dado um conjunto de vetores $\{v_1, v_2, \ldots, v_n\}$ representados como as colunas de uma matriz A, o objetivo do algoritmo de Gram-Schmidt modificado é gerar uma base ortonormal $Q = \{q_1, q_2, \ldots, q_n\}$, onde os vetores $q_1, q_2, \ldots, q_n$ são ortogonais entre si e possuem norma unitária.

O processo inicia com a inicialização, onde o primeiro vetor ortogonal q_1 é simplesmente igual ao vetor original v_1, conforme a equação:

$$q_1 = v_1 \tag{1.4}$$

Para cada vetor v_i, o vetor q_i é calculado subtraindo os componentes de v_i nas direções dos vetores ortogonais $q_1, q_2, \ldots, q_{i-1}$ já calculados. A fórmula para o cálculo de q_i é dada por:

$$q_i = v_i - \sum_{j=1}^{i-1} (q_j \cdot v_i) q_j \tag{1.5}$$

Essa subtração garante que q_i seja ortogonal a todos os vetores $q_1, q_2, \ldots, q_{i-1}$. Após subtrair as projeções, o vetor q_i é normalizado pela equação:

$$q_i = \frac{q_i}{\|q_i\|} \tag{1.6}$$

Caso o vetor q_i seja nulo (ou tenha norma zero), ele é preservado como tal. Esse processo resulta na matriz $Q = [q_1, q_2, \ldots, q_n]$, que contém os vetores ortonormais.

1.4 Implementação em Python

A implementação a seguir realiza a ortogonalização e normalização dos vetores utilizando o algoritmo de Gram-Schmidt modificado.

```
import numpy as np


def gram_schmidt_modified(vectors):
    """
    Aplica o processo de Gram-Schmidt modificado para ortogonalizar um
        conjunto de vetores
    e retorna a matriz Q ortonormal.

    Parâmetros:
        vectors (array-like): Conjunto de vetores a ser ortogonalizado (
            forma (m, n)).

    Retorna:
        Q (ndarray): Matriz ortonormal resultante após a ortogonalização.
    """
    A = np.array(
        vectors, dtype=float)  # Garante que os vetores são do tipo float
    m, n = A.shape
    Q = np.empty((m, n), dtype=float)  # Inicializa Q com o formato correto

    for i in range(n):
      # Inicializa o vetor Q[:, i] com o vetor A[:, i]
      q_i = A[:, i]

```

```
        for j in range(i):
          # Projeta o vetor A[:, i] sobre Q[:, j] e subtrai essa projeção
          q_j = Q[:, j]
          # Usando o operador @ para o produto escalar
          q_i -= (q_j @ A[:, i]) * q_j

        # Normaliza o vetor resultante
        norm_q_i = np.linalg.norm(q_i)
        if norm_q_i > 0:
          Q[:, i] = q_i / norm_q_i
        else:
          # Caso a norma seja zero, preserva o vetor original (tratamento de
              erro)
          Q[:, i] = q_i

    return Q


# Exemplo de uso
A = np.array([[12., -51., 4.],
              [6., 167., -68.],
              [-4., 24., -41]])

# Calculando apenas a matriz Q (Ortonormal)
Q = gram_schmidt_modified(A)

# Verificando a matriz Q
print("Matriz Q (Ortonormal):")
print(np.round(Q, 2))

# Verificando se Q é ortonormal (Q^T * Q deve ser a identidade)
print("\nVerificando se Q^T * Q é a identidade:")
print(np.round(Q.T @ Q, 2))
```

Código 1.2: Implementação do algoritmo de Gram-Schmidt modificado para ortogonalização e normalização de vetores.

1.4.1 Descrição do Funcionamento do Código

A função `gram_schmidt_modified` realiza a ortogonalização de um conjunto de vetores representados como as colunas de uma matriz A. O processo inicia com a inicialização de q_1, onde o primeiro vetor ortogonal q_1 é simplesmente igual ao vetor original v_1. Em seguida, ocorre a ortogonalização: para cada vetor v_i, o vetor q_i é ajustado para ser ortogonal aos vetores $q_1, \ldots, q_{i-1}$ já ortogonais, subtraindo a projeção de v_i sobre cada vetor q_j. Após cada iteração, o vetor ortogonal q_i passa pela normalização. Por fim, é realizada a verificação de ortonormalidade, calculando o produto

$Q^T Q$, que deve resultar na matriz identidade, confirmando que os vetores $q_1, q_2, \ldots, q_n$ são ortogonais entre si e possuem norma unitária.

1.4.2 Saída Esperada

A saída do código será a matriz Q ortonormal gerada pelo algoritmo, e a verificação se o produto $Q^T Q$ é a matriz identidade. Para a entrada do exemplo:

```
Matriz Q (Ortonormal):
[[ 0.86 -0.39 -0.33]
 [ 0.43  0.9   0.03]
 [-0.29  0.17 -0.94]]

Verificando se Q^T * Q é a identidade:
[[ 1. -0. -0.]
 [-0.  1. -0.]
 [-0. -0.  1.]]
```

1.5 Algoritmo de Gram-Schmidt Modificado para Fatoração QR

A fatoração QR é uma técnica fundamental em álgebra linear que decompõe uma matriz A em duas matrizes: uma matriz ortonormal Q e uma matriz triangular superior R, de forma que $A = Q \cdot R$. O algoritmo de Gram-Schmidt modificado é uma maneira eficiente de calcular essa fatoração.

1.5.1 Definições Matemáticas

A fatoração QR de uma matriz A é dada por:

$$A = Q \cdot R, \tag{1.7}$$

onde Q é uma matriz ortonormal de ordem $m \times n$, cujas colunas são vetores ortogonais e normalizados, e R é uma matriz triangular superior de ordem $n \times n$. O algoritmo de Gram-Schmidt modificado gera as matrizes Q e R de forma iterativa. Inicialmente, as matrizes Q e R são inicializadas com matrizes de zeros. Para cada coluna v_i de A, o vetor q_i é calculado subtraindo as projeções de v_i sobre os vetores ortogonais $q_1, q_2, \ldots, q_{i-1}$ já calculados, e em seguida é normalizado. Os coeficientes de projeção, ou seja, os produtos internos $q_j \cdot v_i$, são armazenados em $R[j, i]$, enquanto a norma de q_i é armazenada na posição $R[i, i]$.

1.6 Implementação em Python

A implementação do algoritmo de Gram-Schmidt modificado para a fatoração QR de uma matriz A é apresentada a seguir.

```python
import numpy as np


def gram_schmidt_modified(vectors):
  """
  Aplica o processo de Gram-Schmidt modificado para ortogonalizar um conjunto de vetores
  e calcular a fatoração QR de uma matriz.

  Parâmetros:
      vectors (array-like): Conjunto de vetores a ser ortogonalizado (forma (m, n)).

  Retorna:
      Q (ndarray): Matriz ortonormal (m, n).
      R (ndarray): Matriz triangular superior (n, n).
  """
  A = np.array(vectors, dtype=float)  # Converte os vetores para float
  m, n = A.shape
  Q = np.zeros((m, n), dtype=float)  # Inicializa Q com zeros
  R = np.zeros((n, n), dtype=float)  # Inicializa R com zeros

  for i in range(n):
    # Inicializa o vetor Q[:, i] com o vetor A[:, i]
    q_i = A[:, i]

    for j in range(i):
      # Calcula os coeficientes de projeção (R_ij)
      R[j, i] = Q[:, j] @ A[:, i]
      q_i -= R[j, i] * Q[:, j]  # Subtrai a projeção do vetor Q[:, i]

    # Normaliza o vetor Q[:, i] e calcula o valor R[i, i]
    norm_q_i = np.linalg.norm(q_i)
    if norm_q_i > 1e-10:  # Evita a divisão por zero
      Q[:, i] = q_i / norm_q_i
      R[i, i] = norm_q_i
    else:
      Q[:, i] = q_i  # Preserva o vetor caso sua norma seja zero
      R[i, i] = 0  # Caso o vetor tenha norma zero, R[i, i] é zero

  return Q, R
```

```python
# Exemplo de uso
A = np.array([[12., -51., 4.],
              [6., 167., -68.],
              [-4., 24., -41]])

# Calculando a fatoração QR
Q, R = gram_schmidt_modified(A)

# Verificando a fatoração QR
print("Matriz Q (Ortonormal):")
print(np.round(Q, 2))
print("\nMatriz R (Triangular Superior):")
print(np.round(R, 2))

# Verificando se A = QR
print("\nVerificando se A = QR:")
A_reconstructed = Q @ R
print(np.round(A_reconstructed, 2))
```

Código 1.3: Implementação do algoritmo de Gram-Schmidt modificado para calcular a fatoração QR de uma matriz.

1.6.1 Descrição do Funcionamento do Código

A função `gram_schmidt_modified` realiza a fatoração QR de uma matriz A utilizando o algoritmo de Gram-Schmidt modificado. O código segue os passos descritos anteriormente. Primeiramente, o processo inicia com a inicialização de Q e R, onde são criadas matrizes de zeros para armazenar os resultados da fatoração. Em seguida, ocorre a ortogonalização: para cada vetor v_i da matriz A, o vetor q_i é ortogonalizado subtraindo os componentes de v_i nas direções dos vetores $q_1, q_2, \ldots, q_{i-1}$ já calculados. Após a ortogonalização, realiza-se a normalização do vetor q_i, e sua norma é armazenada na matriz R. Por fim, o algoritmo retorna as matrizes Q e R, que representam a fatoração QR de A.

1.6.2 Saída Esperada

A saída do código será as matrizes Q e R, e a verificação de que $A = Q \cdot R$, como mostrado abaixo:

```
Matriz Q (Ortonormal):
[[ 0.86 -0.39 -0.33]
 [ 0.43  0.9   0.03]
 [-0.29  0.17 -0.94]]

Matriz R (Triangular Superior):
[[ 14.  21. -14.]
 [  0. 175. -70.]
```

```
 [  0.    0.   35.]]

Verificando se A = QR:
[[ 12. -51.    4.]
 [  6. 167. -68.]
 [ -4.  24. -41.]]
```

Capítulo 2

Conclusão

Os algoritmos de Gram-Schmidt, tanto na versão clássica quanto na versão modificada, são amplamente utilizados para a ortogonalização de vetores e para a fatoração QR de matrizes. A fatoração QR é uma ferramenta essencial em álgebra linear, com diversas aplicações, incluindo a solução de sistemas lineares, cálculo de determinantes e análise de estabilidade de sistemas dinâmicos.

A versão clássica do algoritmo realiza a ortogonalização através de projeções sucessivas, enquanto a versão modificada oferece uma abordagem mais eficiente, realizando as projeções de forma direta e evitando a subtração repetida de componentes ortogonais. Sendo a versão modificada, a que permite a decomposição de uma matriz A em duas matrizes: Q, uma matriz ortonormal, e R, uma matriz triangular superior, o que simplifica a resolução de sistemas lineares e facilita a análise da matriz original.

Embora ambos os métodos sejam eficazes, a versão modificada é mais estável numericamente, especialmente em casos com vetores mal condicionados. A fatoração QR, derivada desses algoritmos, é crucial para diversas aplicações em ciência e engenharia, demonstrando a importância dos métodos de ortogonalização na prática. Ambos os algoritmos oferecem soluções robustas e são amplamente utilizados em diversos contextos, com a versão modificada oferecendo vantagens adicionais em termos de precisão e desempenho computacional.