MAP3121 - Métodos Numéricos e Aplicações
Escola Politécnica da Universidade de São Paulo

Exercício Programa 1

Autovalores e Autovetores de Matrizes Tridiagonais Simétricas

Gabriel Macias de Oliveira, NUSP 11260811
Rodrigo Ryuji Ikegami, NUSP 10297265

São Paulo,
2021.

Sumário

1 Introdução

1.1 Descrição do Problema

1.2 Ferramentas Utilizadas

1.3 Execução dos *Scripts*

2 Implementação

2.1 As Rotações de Givens

Conforme [1], as *Rotações de Givens* são transformações lineares ortogonais $Q : \mathbb{R}^n \to \mathbb{R}^n$ da forma $Q(i, j, \theta)$ que operam sobre o espaço de matrizes como uma rotação no plano gerado pelas coordenadas i e j. Dada uma matriz de interesse A, se $Y = Q(i, j, \theta)A$, então:

$$y_{k,l} = \begin{cases} a_{k,l} & k \neq i, j \\ ca_{k,l} - sa_{j,l} & k = i \\ sa_{k,l} + ca_{j,l} & k = j \end{cases} \tag{1}$$

sendo $c = \cos\theta$ e $s = \sin\theta$. Se A é tridiagonal simétrica, podemos, particularmente, explorar essa operação para anular as entradas abaixo da diagonal principal. Se A_n é definido pelos vetores $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n)$ e $\beta = (\beta_1, \beta_2, \cdots, \beta_{n-1})$, ou seja,

$$A = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix}$$

faz-se, iterativamente, $n-1$ rotações com o objetivo de transformar A em uma matriz triangular superior R. Desta forma, para a k-ésima iteração, definimos $Q_k = Q(k, k+1, \theta_k)$ onde θ_k é o ângulo que permite anular a entrada β_k através da rotação. Podemos encontrar tal θ_k de modo numericamente estável fazendo:

$$\tau_k = -\beta_k/\alpha_k \quad c_k = 1/\sqrt{1+\tau_k^2} \quad s_k = c_k\tau_k$$

se $|\alpha_k| > |\beta_k|$ e

$$\tau_k = -\alpha_k/\beta_k \quad s_k = 1/\sqrt{1+\tau_k^2} \quad c_k = s_k\tau_k$$

caso contrário.

Observamos que a aplicação de sucessivas rotações geram novas entradas acima da sobrediagonal, entretanto, embora tais valores existam, não serão calculados, dado que não são necessários para a execução do algoritmo nem para encontrar autovalores e autovetores da matriz A.

Sumarizando o que se discutiu, as sucessivas iterações produzem $R = Q_{n-1} \cdots Q_2 Q_1 A$. Da inversibilidade das rotações de Givens, podemos fazer:

$$A = (Q_1^{-1} Q_2^{-1} \cdots Q_{n-1}^{-1})R$$

Da ortogonalidade da transformação, se escreve: $(Q_1^{-1} Q_2^{-1} \cdots Q_{n-1}^{-1}) = (Q_1^T Q_2^T \cdots Q_{n-1}^T) = Q$. Assim,

$$A = QR$$

2.1.1 Representação das Matrizes em Código

Nesta implementação, lidamos em todas as etapas com Matrizes Tridiagonais Simétricas, conforme já descrito. Uma implementação inicial pode partir de operações utilizando representações puramente matriciais, porém se trabalharmos com matrizes $n \times n$, teremos n^2 posições de memória ocupadas, das quais a maior parte vale 0. Desta forma, estamos desperdiçando memória e tempo (quando consideramos que também iteramos sobre tais zeros).

Destarte, todas as matrizes nesta implementação, à exceção da matriz de autovetores, serão representadas por dois vetores, `alphas`, que armazena as entradas da diagonal principal, e `betas`, que armazena as entradas da sobrediagonal (e, portanto, da subdiagonal também, quando simétrica). Desta seção em diante, nos referenciaremos livremente a tais vetores.

2.1.2 Função de Implementação da Fatoração QR

Com as considerações acima, criou-se a função `qr_factorization`. Dada uma matriz de entrada A, representada por seus vetores `alphas` e `betas`, retorna sua fatoração QR.

As matrizes Q e R da fatoração são representadas por uma 4-tupla ordenada. As posições 0 e 1 da 4-tupla contêm os valores de c_k e s_k das rotações de Givens. Igualmente, as posições 2 e 3 armazenam a diagonal principal e a sobrediagonal da matriz R na forma de `alphas` e `betas`.

A implementação está no Código 2.1.2, abaixo, do qual se retiraram os comentários, mantidos no arquivo original do *script*.

```
1  def qr_factorization(alphas : np.array, betas : np.array) → Tuple[np.array, np.array, np.array, np.array]:
2      c_ks, s_ks = [], []
3      (alphas, betas) = (alphas.copy(), betas.copy())
4
5      for k in range(len(alphas) - 1):
6          if abs(alphas[k]) > abs(betas[k]):
7              tau_k = - betas[k] / alphas[k]
8              c_ks.append(1 / np.sqrt(1 + tau_k**2))
9              s_ks.append(tau_k * c_ks[k])
10         else:
11             tau_k = - alphas[k] / betas[k]
12             s_ks.append(1 / np.sqrt(1 + tau_k**2))
13             c_ks.append(tau_k * s_ks[k])
14
15         alphas[k] = c_ks[k] * alphas[k] - s_ks[k] * betas[k]
16         betas[k] *= c_ks[k - 1] if k > 0 else 1
17         (alphas[k + 1], betas[k]) = (s_ks[k] * betas[k] + c_ks[k] * alphas[k + 1], c_ks[k] * betas[k] - s_ks[k] * alphas[k + 1])
18
19     return (c_ks, s_ks, alphas, betas)
```

Código 2.1.2: Função que implementa a Fatoração QR de uma matriz dada por seus vetores `alphas` e `betas`.

O código segue a descrição formal apresentada anteriormente. As linhas 6 a 13 calculam os valores de τ_k, c_k e s_k. As linhas 15 a 17 correspondem à atualização das linhas da matriz a partir dos valores da diagonal principal e sobrediagonal. Nota-se que na linha 17 utilizou-se a atribuição simultânea da linguagem para permitir a atualização dos valores de α_{k+1} e β_k sem a introdução de uma variável adicional, uma vez que a atualização delas é interdependente e, se feita em sequência, não apresentaria o valor correto.

Uma vez construída a fatoração QR, implementa-se o Algoritmo QR com deslocamento espectral, o que se fará em sequência.

2.2 O Algoritmo QR

Conforme [1], o *Algoritmo QR* determina os autovalores de uma matriz simétrica $A \in \mathbb{R}^{n \times n}$ e é dado por

Vale notar que $A^{(k+1)} = R^{(k)}Q^{(k)} = (Q^{(k)})^T Q^{(k)} R^{(k)} Q^{(k)} = (Q^{(k)})^T A^{(k)} Q^{(k)}$. Disso podemos dizer que $A^{(k+1)}$ e $A^{(k)}$ são (ortogonalmente) semelhantes. Isso significa que, se $A^{(k)}$ é tridiagonal simétrica, então $A^{(k+1)}$ também o é.

Isso significa que podemos fatorar a matriz de entrada, tridiagonal simétrica, e restaurar uma matriz, também tridiagonal simétrica, de maneira iterativa, de modo a convergir a uma matriz diagonal com os autovalores da matriz de entrada.

2.2.1 Função de Atualização da Matriz

Para se implementar o Algoritmo QR, é necessária, além da fatoração QR da matriz A, sua reconstrução, ao fazer o produto RQ, de modo a criar convergência à matriz diagonal dos autovalores da matriz.

Assim, criou-se a função `update_matrix`. Dadas as matrizes Q e R oriundas da fatoração da matriz A, representadas pela 4-tupla ordenada, com a mesma ordem da saída de 2.1.2, retorna a matriz A, reconstruída pela aplicação das rotações reversas a R.

A saída é representada por uma dupla de vetores, `alphas` e `betas`, que armazenam sua diagonal principal e sua sobrediagonal, respectivamente.

A implementação está no Código 2.2.1, abaixo, do qual se retiraram os comentários, mantidos no arquivo original do *script*.

```
1  def update_matrix(c_ks : np.array, s_ks : np.array, alphas : np.array, betas : np.array) → Tuple[np.array, np.array]:
2      (alphas, betas) = (alphas.copy(), betas.copy())
3
4      for i, (c, s) in enumerate(zip(c_ks, s_ks)):
5          (alphas[i], betas[i], alphas[i + 1]) = (c * alphas[i] - s * betas[i], -s * alphas[i + 1], c * alphas[i + 1])
6
7      return (alphas, betas)
```

Código 2.2.1: Função que implementa a reconstrução de uma matriz fatorada, dada pelos vetores `alphas` e `betas` da matriz R e pelos vetores `c_ks` e `s_ks`, que representam os senos e cossenos que compõem a matriz Q.

O código atualiza, na linha 5, os valores das colunas da matriz a partir dos valores dos senos e cossenos das rotações de Givens. Foi utilizada atribuição simultânea das variáveis, na mesma linha, mas a atribuição sequencial dos valores, da esquerda para a direita, também funciona.

Na linha 4, utilizam-se duas funções da linguagem, `enumerate` e `zip`. A função `zip` recebe dois ou mais vetores e retorna outro vetor com os elementos dos vetores pareados em uma n-tupla, cujo comprimento é equivalente ao menor dos vetores. A função `enumerate` recebe um vetor e retorna outro vetor cujos elementos são duplas (índice, elemento) do vetor passado.

2.2.2 Função de Atualização dos Autovetores

2.3 A Heurística de Wilkinson

2.3.1 Função Sinal

2.3.2 Função de Cálculo dos Coeficientes de Deslocamento

2.4 O Algoritmo QR com Deslocamento Espectral

Para o *Algoritmo QR*, podemos acelerar sua convergência ao utilizar os Coeficientes de Deslocamento, pela Heurística de Wilkinson, para alterar os valores da diagonal principal, de tal modo que, ao efetuar uma iteração do algoritmo QR sobre ela, a nova matriz será numericamente mais próxima da matriz Λ desejada.

[PSEUDOCÓDIGO]

Todas as deduções feitas para o *Algoritmo QR* sem deslocamento espectral valem. Ou seja, ao introduzir o deslocamento espectral, não há grandes modificações a serem feitas à implementação do algoritmo para possibilitar o deslocamento.

2.4.1 Função de Implementação do Algoritmo

Finalmente, foi criada a função `qr_algorithm`. Dada uma matriz de entrada, representada pelos vetores `alphas` e `betas`, retorna uma 4-tupla com a matriz calculada após se atingir um determinado erro `epsilon`, além da matriz V, com seus autovetores, e o número de iterações necessárias para se atingir o critério de parada. Além disso, a função recebe um booleano `spectralShift`, que indica se o algoritmo deve ser efetuado com ou sem deslocamento espectral.

A implementação está no Código 2.1.2, abaixo, do qual se retiraram os comentários, mantidos no arquivo original do *script*.

```
1  def qr_algorithm(alphas : np.array, betas : np.array, spectralShift : bool = True, epsilon : float = 1e-6) → Tuple[np.array, np.ar
2      alphas_k = alphas.copy()
3      betas_k = betas.copy()
4      V = np.identity(len(alphas_k))
5      mu = 0
6      iterations = 0
7      for m in reversed(range(1, len(alphas))):
8          while abs(betas_k[m - 1]) ⩾ epsilon:
9              (c_ks, s_ks, alphas_sub, betas_sub) = qr_factorization(alphas_k[: m + 1] - mu * np.ones(m + 1), betas_k[: m + 1])
```

```
10              (alphas_k[: m + 1], betas_k[: m + 1]) = update_matrix(c_ks, s_ks, alphas_sub, betas_sub)
11
12              alphas_k[: m + 1] += mu * np.ones(m + 1)
13
14              V = update_eigenvectors(V, c_ks, s_ks)
15
16              mu = wilkinson_h(alphas_k[: m + 1], betas_k[: m + 1]) if spectralShift else 0
17
18              iterations += 1
19
20      return (alphas_k, betas_k, V, iterations)
```

Código 2.4.1: Função que implementa o Algoritmo QR, com ou sem deslocamento espectral, a partir de uma matriz dada por seus vetores `alphas` e `betas`, até atingir um certo erro `epsilon`.

O código segue a descrição formal apresentada anteriormente. Na linha 9 é executada a fatoração QR da submatriz cujos valores ainda não convergiram. Na linha 10, são atualizados os valores da submatriz, de acordo com a função `update_matrix`. Na linha 12 é desfeito o deslocamento espectral. Na linha 14 é atualizada a matriz dos autovalores. Na linha 16 é calculado μ_{k+1}, o coeficiente da próxima iteração.

3 Construção dos Testes

3.1 Teste 1: Verificação do Algoritmo

3.1.1 Implementação do Teste

3.2 Teste 2: Sistema Massa-Mola com 5 Massas

3.2.1 Implementação do Teste

3.3 Teste 3: Sistema Massa-Mola com 10 Massas

3.3.1 Implementação do Teste

4 Resultados e Discussão

Referências

[1] MAP3121. **EP1: Autovalores e Autovetores de Matrizes Tridiagonais Simétricas: O Algoritmo QR**. Disponível em: <https://edisciplinas.usp.br/pluginfile.php/6249061/mod_resource/content/3/ep1_2021.pdf>. Acesso em: 20 jun. 2021.