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:

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

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

Em conformidade com a discussão de [1], o *Algoritmo QR* determina os autovalores e autovetores de uma matriz $A \in \mathbb{R}^{n \times n}$ e é construído a partir do pseudocódigo 2.2 abaixo, cujos retornos são a matriz A em sua forma diagonalizada Λ e sua matriz de autovetores V, em que se armazena os autovetores nas respectivas colunas.

Algorithm 1 Algoritmo QR

$A^{(0)} = A$
$V^{(0)} = I_n$
for $k = 1, 2, ...$ **do**
    $A^{(k)} \to Q^{(k)} R^{(k)}$
    $A^{(k+1)} = R^{(k)} Q^{(k)}$
    $V^{(k+1)} = V^{(k)} Q^{(k)}$
end for

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.

Consequentemente 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. Iremos, nos próximos parágrafos, explorar essa operação e como implementá-la[1].

Recuperando a definição de Q, temos $Q = (Q_1^T Q_2^T \cdots Q_{n-1}^T)$, assim $RQ = R(Q_1^T Q_2^T \cdots Q_{n-1}^T)$ que podemos reescrever como $RQ = \left\{(Q_1^T Q_2^T \cdots Q_{n-1}^T)^T R^T\right\}^T = \left\{Q_{n-1} \cdots Q_2 Q_1 R^T\right\}^T$. Logo, a reconstrução da matriz A é a aplicação das rotações de Givens às colunas de R.

Portanto, recuperando a definição de 1, podemos analisar a operação sob mesma perspectiva, resultando, portanto nas considerações da relação 2 abaixo. Se $Y = AQ(i, j, \theta)^T$, vale:

$$y_{k,l} = \begin{cases} a_{k,l} & k \neq i, j \\ ca_{k,i} - sa_{k,j} & l = i \\ sa_{k,i} + ca_{k,j} & l = j \end{cases} \tag{2}$$

Logo, computamos RQ iterativamente aplicando a rotação $Q_k^T = Q^T(k, k+1, \theta_k)$ a cada passo em que θ_k é originado do par `(c_k, s_k)` de cossenos e senos definido previamente pela representação da matriz Q da fatoração de A.

É válido notar que as parcelas em $R_{k,k+2}$ produzidas pelas rotações de Givens originais não influenciam nos cálculos em 2, haja vista que as entradas da sobrediagonal $\beta' = (\beta'_1, \beta'_2, \cdots, \beta'_{n-1})$ são gerados pela subdiagonal, por simetria da matriz, justificando a ausência de um vetor $\gamma = (\gamma_1, \cdots, \gamma_{n-2})$ para as armazenar.

A partir dessa descrição, 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 de entrada.

[1] **Observação:** por brevidade, nos trechos desta descrição, omitiremos os índices de iteração das operações matriciais, que ficarão subentendidos.

2.2.2 Função de Atualização dos Autovetores

O mesmo desenvolvimento a respeito da operação $A^{(k+1)} = R^{(k)}Q^{(k)}$ na seção anterior vale para a operação $V^{(k+1)} = V^{(k)}Q^{(k)}$ e, por causa disso, a implementação desta operação é praticamente análoga à primeira. A matriz $Q^{(k)} = (Q_1^T Q_2^T \cdots Q_{n-1}^T)^{(k)}$ aplicada à direita atua como uma série de rotações de Givens sobre as colunas de $V^{(k)}$. Apesar disso, devemos ter em mente que a matriz de autovalores *não é tridiagonal simétrica*. Por conseguinte, devemos operar sobre a matriz V pura, conforme 2, sem considerar a abstração em `alphas` e `betas` como feito para as demais matrizes.

É com essa constatação que se desenvolveu a função `update_eigenvectors`, cuja implementação está no Código 2.2.2 abaixo, do qual omitimos os comentários que estão no *script* original.

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

Código 2.2.2: Função que implementa a atualização dos autovetores de A, armazenando-os nas colunas de V.

As entradas da função são as matrizes V e Q da `k`-ésima iteração, esta última representada por seus vetores `c_ks` e `s_ks` de cossenos e senos. Na linha 4, executamos um laço sobre as colunas de V, associando a cada coluna i o par (c_i, s_i) da rotação $Q(i, i+1, \theta_i)^T$. A rotação de Givens é executada na linha 5. A função retorna a matriz $V^{(k+1)}$, contendo os autovetores atualizados até a iteração atual do algoritmo QR.

2.3 O Algoritmo QR com Deslocamento Espectral

2.3.1 A Heurística de Wilkinson

De acordo com [1], a taxa de convergência do Algoritmo QR depende da razão $|\lambda_{j+1}/\lambda_j|$, o que a torna muito lenta caso a razão entre os módulos de autovalores consecutivos esteja próxima de 1.

Visando acelerar a convergência do método, podemos subtrair da matriz $A^{(k)}$, em cada iteração, a matriz identidade múltiplicada por uma constante escalar μ_k, denominada *constante de deslocamento espectral*, que esteja próxima a um autovalor.

É importante perceber que μ_k deve ser recalculado a cada iteração. Seguindo o argumento de [1], ao alterarmos o pseudocódigo 2.2, fazendo iterações da forma

$$A^{(k)} - \mu_k I_n \to Q^{(k)} R^{(k)}$$

$$A^{(k+1)} = R^{(k)} Q^{(k)} + \mu_k I_n$$

a taxa de convergência de uma entrada da diagonal principal $\alpha_j^{(k)}$ para o respectivo autovalor λ_j será proporcional a $|(\lambda_j - \mu_k)/(\lambda_{j-1} - \mu_k)|$. Se μ_k está próximo de λ_j, esperamos que $\beta_{j-1}^{(k)} \to 0$ mais rápido que $\beta_i^{(k)}, \forall i < j$, o que corresponde a $\alpha_j^{(k)}$ convergir rapidamente para λ_j.

Para calcular os coeficientes de deslocamento μ_k de cada iteração, utilizamos a heurística de Wilkinson.

Heurística de Wilkinson: Seja $d_k = (\alpha_{n-1}^{(k)} - \alpha_n^{(k)})/2$, definimos $\mu_k = \alpha_n^{(k)} + d_k - sgn(d_k)\sqrt{d_k^2 + (\beta_{n-1}^{(k)})^2}$, onde $sgn(d)$ é a função *sinal*, isto é:

$$sgn(d) = \begin{cases} 1, & d \geq 0 \\ 0, & d < 0 \end{cases} \tag{3}$$

2.3.2 Função Sinal

A implementação em código da função `sgn` é imediata, como se observa no código 2.3.2 abaixo, do qual se retiraram os comentários, mantidos no *script* original.

```
def sgn(x):
    return copysign(1, x)
```

Código 2.3.2: Função *sinal* da heurística de Wilkinson.

A função `copysign(x, y)`, da biblioteca `math` retorna um número construído pelo módulo de `x` e o sinal de `y`, assim a função `sgn` recebe um $x \in \mathbb{R}$ e copia o sinal de x para o número 1, resultando no mesmo efeito descrito pela equação 3.

2.3.3 Função de Cálculo dos Coeficientes de Deslocamento

A função exibida no Código 2.3.3 implementa o cálculo dos coeficientes de deslocamento μ_k a partir da heurística de Wilkinson.

```
def wilkinson_h(alphas : np.array, betas : np.array) -> float:
    d_k = (alphas[len(alphas) - 1]  - alphas[len(alphas) - 2]) / 2
    return alphas[len(alphas) - 1] + d_k - sgn(d_k) * np.sqrt(d_k**2 + betas[len(alphas) - 2]**2)
```

Código 2.3.3: Função de cálculo dos coeficientes de deslocamento pela heurística de Wilkinson.

Todas as deduções feitas para o *Algoritmo QR* sem deslocamento espectral valem. Isto é, todas as matrizes $A^{(k)}$ são ortogonalmente semelhantes a A e, ao introduzir o deslocamento espectral, as hipóteses realizadas anteriormente para particularização do algoritmo a Matrizes Tridiagonais Simétricas ainda são verificadas.

Fixado um ϵ de tolerância, consideramos a convergência de $\alpha_n^{(k)}$ para λ_n quando $|\beta_{n-1}^{(k)}| < \epsilon$. Já determinado o autovalor associado a `n`-ésima posição, continua-se a execução do algoritmo com a submatriz tridiagonal $n-1 \times n-1$ obtida pelo *slice* de A. Esta rotina se repete até obtermos todos os autovalores de A mediante a tolerância fornecida.

2.3.4 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.array, np.array, int]:
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 2.3.4: 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 7 é criada uma variável `m`, que vai de $N-1$ até 1, utilizada para determinar o tamanho da submatriz na iteração atual. A cada iteração, a matriz sobre a qual os cálculos são feitos é menor que a anterior, de tal forma que, quando um determinado β_j converge, seu valor não é mais calculado nas próximas iterações. A verificação da condição de convergência é feita na linha 8, ao se comparar o β_j mais à direita, na submatriz atual, com um dado ϵ. 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

Na seção 2.3 a) de [1], é apresentada uma família de matrizes cujos autovalores e autovetores são conhecidos. É pedido que se execute o Algoritmo QR sobre algumas dessas matrizes para testar o funcionamento da implementação feita. Além disso, é pedido para que se compare o número de iterações necessárias para a convergência

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.