MAP3121 - Métodos Numéricos e Aplicações

Escola Politécnica da Universidade de São Paulo

Exercício Programa 2

Autovalores e Autovetores de Matrizes Reais Simétricas

Gabriel Macias de Oliveira, NUSP 11260811, Eng. Elétrica

Rodrigo Ryuji Ikegami, NUSP 10297265, Eng. Elétrica

São Paulo,

2021.

Sumário

1 Introdução **2**

1.1 Ferramentas Utilizadas . . . 2

1.2 Execução dos *Scripts* . . . 2

2 Implementação **3**

2.1 As Transformações de Householder . . . 3

2.1.1 Função de Implementação da Tridiagonalização de uma Matriz Real Simétrica . . 4

2.2 O Algoritmo QR . . . 6

2.3 Leitura de Matrizes em Arquivos . . . 6

2.4 Leitura de Treliças em Arquivos . . . 7

3 Construção dos Testes **9**

3.1 Teste A: Matriz de Autovalores Inteiros Conhecidos . . . 9

3.2 Teste B: Matriz de Autovalores dados por Fórmula . . . 10

3.3 Aplicação do Algoritmo a Treliças Planas . . . 10

3.4 Função Principal . . . 10

Referências **11**

1 Introdução

1.1 Ferramentas Utilizadas

Foram utilizadas as seguintes ferramentas para construção do código:

- Linguagem de Programação: *Python 3.7.9+*
- Bibliotecas Externas:
  - `numpy`, para trabalhar com aritmética de vetores
  - `matplotlib`, para produção de gráficos e animações
- *IDE*: *Visual Studio Code*
- Desenvolvimento Paralelo: *Git*

Além das bibliotecas externas, utilizaram-se as bibliotecas nativas `math`, para funções matemáticas básicas, `typing`, para utilizar tipos estáticos em *Python*, e `sys` para personalização da CLI.

Todos os testes em que são envolvidas métricas de tempo / número de iterações foram executados com base em um AMD Ryzen 5 3600X @ 4.2 GHz, portanto sendo suscetíveis a variações.

Todo o código está concentrado no arquivo `main.py`, cujos detalhes de execução se encontram em sequência e no arquivo `LEIA-ME.txt`.

Este relatório foi tipografado em LaTeX.

1.2 Execução dos *Scripts*

Estando o *Python* atualizado para uma versão compatível, isto é, 3.7.9 ou mais recente, deve-se certificar que ambas bibliotecas `numpy` e `matplotlib` estejam instaladas. Caso contrário, basta executar `pip install -r requirements.txt` em um terminal, para recebê-las.

O arquivo principal deve ser executado no mesmo diretório em que foi descompactado, utilizando o comando `python main.py`. A exibição do terminal deve ser da CLI que acompanha o programa, conforme a Figura **??**.

2 Implementação

2.1 As Transformações de Householder

Conforme [@MAP3121], as *Transformações de Householder* são transformações lineares ortogonais H_w : $\mathbb{R}^n \to \mathbb{R}^n$ da forma $H_w = I - \frac{2ww^T}{w \cdot w}$ que operam sobre o espaço de vetores como uma reflexão em relação ao espaço $w^\perp$. Dado um vetor de interesse x, se $y = H_w x$, então:

$$y = x - 2\frac{w \cdot x}{w \cdot w} w \,.$$

Dados dois vetores x e y, não nulos em $\mathbb{R}^n$, é possível definir uma transformação de Householder tal que $H_w x = \lambda y$, com $\lambda \in \mathbb{R}$. Para tanto basta se definir $w = x \pm \frac{||x||}{||y||} y$. (1)

Esta propriedade se torna extremamente útil para a tridiagonalização de matrizes reais simétricas. Para cada coluna i de uma matriz dada A, podemos definir uma transformação de Householder com:

$$w_i = \tilde{a}_i + \delta \frac{||\tilde{a}_i||}{||e_{i+1}||} e_{i+1} = \tilde{a}_i + \delta ||\tilde{a}_i|| e_{i+1} \,.$$

sendo $\delta = \pm 1$, e_{i+1} o i+1-ésimo versor da base canônica de $\mathbb{R}^n$ e $\tilde{a}_i$ composta pelos elementos da coluna i de A, exceto os pertencentes à diagonal principal e acima dela. Isto é,

$$\tilde{a}_i = (0, 0, ..., 0, A_{i+1,i}, A_{i+2,i}, ..., A_{n-1,i}, A_{n,i})^T \,.$$

De acordo com o proposto em [@MAP3121], utilizamos δ com sinal igual ao de $A_{i+1,i}$ para cada w_i.

Utilizando a propriedade em (1), podemos provar que

$$H_{w_i} \tilde{a}_i = \tilde{a}_i - \delta w_i H_{w_i} \tilde{a}_i = (0, 0, ..., 0, -\delta ||\tilde{a}_i||, 0, ..., 0)^T \,.$$

Ou seja, a coluna i, após a transformação H_{w_i}, possui como único elemento não nulo o módulo de $\tilde{a}_i$ na posição i, com sinal oposto ao de $A_{i+1,i}$.

Assim, temos que

$$H_{w_1} A = \begin{bmatrix} x & x & x & ... & x \\ x & x & x & ... & x \\ 0 & x & x & ... & x \\ ... & ... & ... & ... & ... \\ 0 & x & x & ... & x \end{bmatrix}$$

onde os x representam valores quaisquer.

E, como H_{w_1} e A são simétricas, podemos fazer

$$H_{w_1} A H_{w_1} = \begin{bmatrix} x & x & 0 & ... & 0 \\ x & x & x & ... & x \\ 0 & x & x & ... & x \\ ... & ... & ... & ... & ... \\ 0 & x & x & ... & x \end{bmatrix}$$

para zerar os elementos à direita da sobrediagonal na primeira coluna.

A matriz resultante, pelo mesmo motivo, também é simétrica. Assim, podemos aplicar sucessivas transformações de Householder à direita e à esquerda de A de forma a obter uma matriz semelhante a A, T, tridiagonal.

Vale ressaltar que, para cada H_{w_i} multiplicado à esquerda de A, apenas as linhas $i+1$ a n têm seus elementos modificados. Simetricamente, quando se multiplica H_{w_i} à direita de A, apenas as colunas $i+1$ a n têm seus elementos modificados. (2)

Ao fim das transformações, obteremos a expressão $T = HAH^T$, com:

$$H^T = H_{w_1} H_{w_2} ... H_{w_{n-1}} H_{w_n} .$$

Para economizar tempo e memória, definimos $\bar{w}_i$ e $\bar{a}_i$, que são equivalentes a w_i e $\tilde{a}_i$, respectivamente, mas sem os i primeiros valores, pois são todos zero.

2.1.1 Função de Implementação da Tridiagonalização de uma Matriz Real Simétrica

Feitas as considerações acima, criamos a função `tridiagonalization`, que recebe uma matriz simétrica `A` como entrada e devolve a matriz T tridiagonal, representada por dois vetores, `alphas` e `betas`, que representam sua diagonal principal e sua sobrediagonal, respectivamente, e a matriz `Ht`, que representa H^T, descrita anteriormente.

A implementação feita se utilizada das propriedades descritas em (1) e (2) para aumentar a eficiência do código. Em cada iteração, podemos trabalhar sobre uma submatriz de A, sobre a qual faremos as contas, já que valores fora dela não são modificados, exceto a coluna/linha cuja maioria dos valores serão zerados. Os único valor que não são zero pertencem à diagonal principal e sua sobre/subdiagonal. Para os valores da diagonal, basta tomar $A_{i,i}$ na iteração i, pois seu valor não é afetado por nenhuma das transformações de Householder subsequentes. Para os valores da sobrediagonal, basta tomar $-\delta||\tilde{a}_i||$, como demonstrado anteriormente.

A implementação está no Código 2.1.1, abaixo, do qual se retiraram os comentários, mantidos no arquivo original do *script*.

```
def tridiagonalization(A: np.array) -> Tuple[np.array, np.array, np.array]:
```

```
2       A = A.copy()
3       alphas = []
4       betas = []
5
6       H = np.identity(np.size(A, 0))
7
8       for m in reversed(range(2, np.size(A, 0))):
9           w_i = A[1:, 0]
10
11          alphas.append(A[0, 0])
12          betas.append(-sgn(w_i[0]) * np.sqrt(np.dot(w_i, w_i)))
13
14          w_i[0] -= betas[-1]
15          w_i2 = np.dot(w_i, w_i)
16
17          A = A[1:, 1:]
18
19          for Acol, Arow, Hrow in zip(np.transpose(A), A, H[:, -m:]):
20              Acol -= 2 * np.dot(w_i, Acol) / w_i2 * w_i
21              Arow -= 2 * np.dot(w_i, Arow) / w_i2 * w_i
22              Hrow -= 2 * np.dot(w_i, Hrow) / w_i2 * w_i
23
24      alphas.extend(np.diag(A))
25      betas.append(A[1, 0])
26
27      return (np.array(alphas), np.array(betas), H)
```

Código 2.1.1: Função que implementa a tridiagonalização de uma matriz dada, `A`.

O código segue a descrição formal apresentada anteriormente. A linha 9 define o vetor $\bar{w}_i$ da Tranformação de Householder, $H_{\bar{w}_i}$, de uma dada iteração `i` e o inicializa com $\bar{a}_i$. Sa linhas 11 e 12 adicionam os elementos calculados da diagonal principal e da sua sobrediagonal aos vetores `alphas` e `betas`, respectivamente. A linha 14 modifica o w_i de acordo com a expressão $\bar{w}_i = \bar{a}_i + ||\bar{a}_i||\delta e_1$. A linha 15 define uma variável auxiliar `w_i2`, equivalente a $\bar{w}_i \cdot \bar{w}_i$. A linha 17 atualiza a variável `A` para armazenar a submatriz de uma dada iteração. As linhas 19 a 22 executam as multiplicações $H_{\bar{w}_i}\bar{A}H_{\bar{w}_i}$ e $H^T H_{\bar{w}_i}$. As linhas 24 e 25 adicionam os últimos elementos da diagonal principal e da sobrediagonal da matriz resultante em `alphas` e `betas`, respectivamente.

2.2 O Algoritmo QR

Após se obter a matriz T, tridiagonal, a partir das transformações de Householder, podemos obter seus autovetores e autovalores utilizando o *Algoritmo QR*. Apesar de A e T serem matrizes semelhantes, isto é, possuem mesmos autovalores, seus autovetores são distintos. Ou seja, para se obter os autovalores de A, precisamos que $V^{(0)}$ seja equivalente a H^T, pois, utilizando-se a matriz identidade como $V^{0)}$, obteríamos os autovetores de T.

Para a implementação do Algoritmo QR, foi utilizada a mesma função, `qr_algorithm`, do EP anterior. A única modificação feita sobre ela foi a adição de um parâmetro de entrada, `V0`, que é utilizado ao invés da identidade para o cálculo dos autovetores.

2.3 Leitura de Matrizes em Arquivos

Ambos testes A e B podem ter suas matrizes de entrada obtidas a partir da leitura de um arquivo, conforme detalhado em [@MAP3121]. Neste arquivo, que utilizamos como padrão neste exercício-programa, a primeira linha contém o tamanho `n` da matriz $A \in \mathbb{R}^{n \times n}$. As linhas subsequentes contêm as entradas equivalentes de cada linha da matriz, sendo as entradas separadas por espaços, uma linha da matriz por linha do arquivo. O código 2.3 implementa essa função.

```python
1  def matrix_from_file(filename):
2      with open(filename, encoding="utf-8") as file:
3          matrix_size: int = int(file.readline())
4          matrix = np.zeros((matrix_size, matrix_size))
5
6          treatline = lambda line: list(map(float, line.split()))
7          rows = list(filter(lambda line: len(line) > 0, map(treatline, file.readlines())))
8
9          for i, line in enumerate(rows):
10             matrix[i, :] = line
11
12     return matrix
```

Código 2.3: Função de leitura de uma matriz a partir de um arquivo.

Em resumo, abrimos os arquivos e extraímos o tamanho da matriz pelo valor (inteiro) da primeira linha. Criamos uma matriz preenchida com zeros do tamanho lido. Funcionalmente, transformamos cada linha de uma *array* de *strings* para uma *array* de *floats*, por meio da aplicação de dois mapeamentos nas linhas 6 e 7. Aplica-se um filtro que garante que as linhas lidas não são vazias, após o qual se converte a lista de *arrays* para uma matriz de retorno.

2.4 Leitura de Treliças em Arquivos

É possível também para a aplicação do algoritmo ao problema de treliças planas, descrever as estruturas para as quais desejamos solucionar por meio de arquivos. Em particular, utilizaremos a descrição dada em [@MAT3121]. Para isso, implementamos duas funções. A primeira, `addBar` é uma função auxiliar que, dados os índices `i` e `j` dos nós que formam uma barra, seu comprimento `L`, o cosseno e seno do ângulo que forma com a horizontal, o módulo de Young em Pa do material da barra, bem como sua densidade `p` em kg/m^3 e a área da seção transversal da barra `A` em m^2, adiciona a contribuição da barra correspondente às matrizes de massa M e de rigidez K, cuja descrição está no código 2.4.

```python
1  def addBar(
2      i: int,
3      j: int,
4      L: float,
5      c: float,
6      s: float,
7      E: float,
8      p: float,
9      A: float,
10     M: np.array,
11     K: np.array,
12 ):
13     mass_contribution = 0.5 * p * A * L
14     M[i] += mass_contribution
15
16     local_stiffness = (A * E) / L * np.array([[c ** 2, c * s], [c * s, s ** 2]])
17     K[2 * i : 2 * i + 2, 2 * i : 2 * i + 2] += local_stiffness
18
19     if j in range(len(M)):
20         M[j] += mass_contribution
21         K[2 * i : 2 * i + 2, 2 * j : 2 * j + 2] += -local_stiffness
22         K[2 * j : 2 * j + 2, 2 * i : 2 * i + 2] += -local_stiffness
23         K[2 * j : 2 * j + 2, 2 * j : 2 * j + 2] += local_stiffness
```

Código 2.4: Função auxiliar que adiciona a contribuição de uma barra às matrizes que descrevem o sistema total.

Na linha 13, calculamos a contribuição da massa da barra para os nós `i` e `j` que a definem. Notamos que `j` pode ser um nó fixado, portanto devemos verificar se este índice corresponde a um ponto móvel, isto é, se j é menor que o tamanho do vetor M. Na linha 16, calculamos a matriz de rigidez local $K_{i,j}$, adicionando essa contribuição à matriz de rigidez total conforme descrito em [@MAT3121].

Considerando que a primeira linha do arquivo contém o número total de nós, o número de nós livres e o número de barras, e que a segunda linha contém a densidade, a área da seção transversal e o módulo de Young (em GPa), bem como as linhas subsequentes descrevem cada barra, cujas entradas são os nós que compõem a barra, o ângulo com a horizontal e o comprimento da barra, nesta ordem, separadas por espaços, criou-se a função 2.4 que implementa a leitura de uma treliça por um arquivo.

```
1  def truss_from_file(filename):
2      with open(filename, encoding="utf-8") as file:
3          total_nodes, free_nodes, _ = map(int, file.readline().split())
4          p, A, E = map(float, file.readline().split())
5          E *= 1e9
6
7          treatline = lambda line: tuple(map(int, line.split()[:2])) + tuple(
8              map(float, line.split()[2:])
9          )
10         bars = list(
11             filter(lambda line: len(line) > 0, map(treatline, file.readlines()))
12         )
13
14         K = np.zeros((2 * free_nodes, 2 * free_nodes), float)
15         M = np.zeros(free_nodes, float)
16
17         for bar in bars:
18             (i, j, theta, L) = bar
19             theta = np.deg2rad(theta)
20             addBar(i - 1, j - 1, L, np.cos(theta), np.sin(theta), E, p, A, M, K)
21
22     return M, K, total_nodes, free_nodes, bars
```

Código 2.4: Função de leitura de uma treliça plana a partir de um arquivo.

Nas linhas 3 e 4 obtemos os dados da treliça, convertendo o módulo de Young de GPa em Pa pela multiplicação por 10^9 na linha 5. Convertemos as linhas do arquivo de *arrays* de *strings* para *arrays* de *floats* da mesma forma que realizado na última seção. Criamos a matriz de rigidez total K e o vetor de massas M utilizando o tamanho lido no arquivo. Para cada barra lida, adicionamos sua contribuição às matrizes de descrição do sistema nas linhas 18 a 20, convertendo, primeiramente, o ângulo de graus para radianos. Retorna-se o vetor de massa, a matriz de rigidez, o número de nós totais e livres e o número de barras. A razão pela representação de M como uma matriz será descrita em seção posterior.

3 Construção dos Testes

Foram construídos 4 diferentes rotinas de teste para o programa. As duas primeiras são implementações dos testes A e B descritos em [@MAP3121]. Já a terceira corresponde à aplicação para solução de treliças planas em oscilações de baixa energia total. Por fim, a quarta e última rotina permite ao usuário verificar a utilização do algoritmo para uma matriz qualquer, inserida manual- ou automaticamente. Em seguida, descreveremos as construções destes testes, na ordem que foram apresentados.

3.1 Teste A: Matriz de Autovalores Inteiros Conhecidos

Nesta instância, desejamos obter os autovalores e autovetores da matriz A descrita abaixo, cujos valores são conhecidos e valem $\Lambda = (7, 2, -1, -2)$, sendo:

$$A = \begin{bmatrix} 2 & 4 & 1 & 1 \\ 4 & 2 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

Com os autovalores e autovetores calculados, verificamos se é válida a relação $Av_j = \lambda_j v_j$ para cada autovalor λ_j e seu autovetor correspondente, bem como realizar o teste de ortogonalidade dos autovetores, isto é, identificar se vale $VV^T = I$. O Código 3.1 abaixo implementa o teste.

```
1 Código Dahora ;D
```

Código 3.1: Implementação do Teste A.

Na linha 8, lemos a matriz do arquivo `input-a`, fornecido com o enunciado, bem como enviado no arquivo compactado da solução do exercício. Com a matriz lida, executamos o processo de tridiagonalização por transformações de Householder na linha 13, decompondo a matriz em seus vetores de `alphas` e `betas`, de acordo com a descrição do Exercício Programa 1. Na linha 24, aplica-se o Algoritmo QR, cujo resultado contém os autovalores e autovetores da matriz A. Nas linhas 40 e 41, verificamos a definição para os autovalores e vetores encontrados, isto é, se verificamos $Av = \lambda v$. Por fim, a linha 62 executa o teste de ortogonalidade.

3.2 Teste B: Matriz de Autovalores dados por Fórmula

Neste teste, encontraremos os autovalores e autovetores da matriz A abaixo, cujos autovalores são dados pela fórmula $\lambda_j = \frac{1}{2}\left[1 - \cos\frac{(2i-1)\pi}{2n+1}\right]^{-1}$ com $i = 1, 2, \ldots, n$.

$$A = \begin{bmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ n-1 & n-1 & n-2 & \cdots & 2 & 1 \\ n-2 & n-2 & n-2 & \cdots & 2 & 1 \\ \vdots & \vdots & \vdots & \cdots & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Apresenta-se as mesmas verificações descritas no teste acima. O Código 3.2 abaixo detalha a implementação do teste.

```
Código Dahora ;D
```

Código 3.2: Implementação do Teste B.

A implementação é análoga ao Teste A, diferindo apenas na construção dos autovalores para comparação, pois são dados pela fórmula apresentada.

3.3 Aplicação do Algoritmo a Treliças Planas

3.4 Função Principal

A função principal do programa segue uma interface simples, com opções numeradas que o usuário pode selecionar.

Referências