UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE MATEMÁTICA

ANA BÁRBARA OMITA DOS SANTOS

CÁLCULO NUMÉRICO DE AUTOVALORES, DECOMPOSIÇÃO EM VALORES SINGULARES E APLICAÇÕES

SÃO CARLOS - SP
2025

ANA BÁRBARA OMITA DOS SANTOS

CÁLCULO NUMÉRICO DE AUTOVALORES, DECOMPOSIÇÃO EM VALORES SINGULARES E APLICAÇÕES

Monografia apresentada ao curso de "Bacharelado" em Matemática da Universidade Federal de São Carlos.

Orientador(a): Prof. Dr. Sávio Brochini Rodrigues

SÃO CARLOS - SP
2025

Agradecimentos

Agradeço primeiramente a Deus, por me guiar, me dar força e me sustentar em cada passo desta jornada, permitindo que eu chegasse até este momento de conclusão.

Dedico esta conquista, com profundo amor e eterna saudade, ao meu pai. Ele sempre me apoiou incondicionalmente e acreditou em mim, sendo um pilar fundamental para que eu chegasse onde cheguei. Mesmo tendo partido durante a minha graduação, sei que sua força e seu incentivo estiveram comigo até o fim. Esta vitória também é sua, pai.

Ao meu noivo e à minha mãe, minha eterna gratidão. Vocês foram meu porto seguro, com paciência, amor e apoio indispensáveis, especialmente nos momentos mais desafiadores desta caminhada.

Por fim, agradeço aos meus professores, amigos, colegas e a todos os demais envolvidos que, direta ou indiretamente, contribuíram com seu tempo, conhecimento e incentivo para que este trabalho fosse possível.

Resumo

Este trabalho apresenta um estudo dos métodos numéricos utilizados para o cálculo de autovalores de matrizes reais, com foco na combinação entre a redução à forma de Hessenberg e o método QR com *shift* de Wilkinson. Inicialmente, revisam-se os fundamentos teóricos essenciais, incluindo transformações de similaridade, diagonalização, fatoração de Schur e propriedades espectrais. Em seguida, descrevem-se as reflexões de Householder e sua aplicação na triangularização parcial de matrizes, etapa fundamental para acelerar algoritmos iterativos de autovalores. Na segunda parte, analisam-se os métodos iterativos clássicos — potência, potência inversa, potência inversa com *shift* e quociente de Rayleigh — culminando na formulação do método QR com *shift*, juntamente com o mecanismo de deflação. Por fim, as duas fases são implementadas e aplicadas a um exemplo numérico, construído com autovalores pré-definidos e pequenas perturbações. O processo iterativo completo é documentado, evidenciando a rápida convergência do método, a eficiência do *shift* de Wilkinson e a estabilidade da deflação.

Palavras-chave: método da potência inversa; iteração inversa com *shift*; cálculo de autovalores; álgebra linear numérica.

Abstract

This work presents a study of numerical methods used for computing eigenvalues of real matrices, with a focus on the combination of the reduction to Hessenberg form and the QR method with Wilkinson's shift. Theoretical foundations are first reviewed, including similarity transformations, diagonalization, the Schur factorization, and spectral properties. Next, Householder reflections and their role in the partial triangularization of matrices are described, a fundamental step for accelerating iterative eigenvalue algorithms. In the second part, classical iterative methods are analyzed — the power method, inverse power method, inverse iteration with shift, and the Rayleigh quotient iteration — culminating in the formulation of the QR method with shift, together with the deflation mechanism. Finally, the two phases are implemented and applied to a numerical example constructed with predefined eigenvalues and small perturbations. The complete iterative process is documented, highlighting the rapid convergence of the method, the efficiency of Wilkinson's shift, and the stability of deflation.

Keywords: inverse power method; shifted inverse iteration; eigenvalue computation; numerical linear algebra.

Sumário

1	**INTRODUÇÃO**	**8**
2	**FUNDAMENTOS TEÓRICOS DE AUTOVALORES**	**9**
2.1	**Polinômio característico**	**10**
2.2	**Transformações de similaridade**	**11**
2.3	**Determinante e traço**	**13**
2.4	**Matrizes diagonalizáveis**	**14**
2.5	**Diagonalização unitária**	**15**
2.6	**Fatoração de Schur**	**16**
3	**FASE 1: TRIANGULARIZAÇÃO DE MATRIZES**	**18**
3.1	**Reflexões de Householder**	**18**
3.2	**Redução à forma de Hessenberg**	**19**
4	**FASE 2: ITERAÇÃO QR E EXTRAÇÃO DE AUTOVALORES**	**22**
4.1	**Quociente de Rayleigh**	**23**
4.2	**Iteração da potência**	**23**
4.3	**Método da potência inversa com shift**	**25**
4.4	**Método do Quociente de Rayleigh**	**27**
4.5	**Iteração simultânea**	**29**
4.6	**QR sem shifts**	**30**
4.7	**Método QR com Shift**	**35**
4.7.1	Shift de Wilkinson	36
4.8	**Exemplo “fácil”**	**37**
4.9	**Exemplo “difícil”**	**40**
5	**APLICAÇÃO FASE 1 E 2**	**45**
5.1	**Fase 1 — Redução à Forma de Hessenberg**	**45**
5.2	**Fase 2 — Método QR com Shift de Wilkinson e Deflação**	**45**
6	**CONCLUSÃO**	**49**
	REFERÊNCIAS	**50**
A	**CÓDIGO MÉTODO DA POTÊNCIA**	**51**
B	**CÓDIGO MÉTODO DA POTÊNCIA INVERSA**	**53**

C CÓDIGO ITERAÇÃO DO QUOCIENTE DE RAYLEIGH 57
D CÓDIGO QR SEM SHIFTS 61
E CÓDIGO QR COM SHIFTS 65
F EXEMPLO “FÁCIL” 66
G EXEMPLO “DIFÍCIL” 71
H EXEMPLO FASES 1 E 2 78

1 Introdução

O problema de autovalores e autovetores representa um dos pilares da Álgebra Linear. Estes conceitos são fundamentais para a compreensão das transformações lineares, revelando suas propriedades geométricas e estruturais intrínsecas. O estudo de autovalores é frequentemente usado para a análise da estabilidade de sistemas, a diagonalização de matrizes e a decomposição de espaços vetoriais em subespaços invariantes, fornecendo uma base teórica para diversos campos da matemática e suas extensões.

Embora a definição de autovalores através do polinômio característico seja conceitualmente simples, aplicar diretamente algoritmos para o cálculo de suas raízes mostra-se computacionalmente inviável ou numericamente instável para matrizes de grande dimensão. Esta dificuldade impulsionou o desenvolvimento de toda uma área da Álgebra Linear Numérica focada em métodos iterativos, que buscam aproximar os autovalores de forma eficiente e precisa.

O objetivo principal deste trabalho é realizar um estudo detalhado e comparativo dos principais métodos numéricos iterativos para a extração de autovalores. Busca-se não apenas descrever o funcionamento teórico de cada algoritmo, mas também analisar seu comportamento prático e eficiência computacional.

Para alcançar este objetivo, o texto estabelecerá primeiramente os fundamentos teóricos necessários, como as transformações de similaridade e a Fatoração de Schur. Em seguida, serão abordadas técnicas preparatórias essenciais para a eficiência dos métodos, notadamente a redução à forma de Hessenberg através das Reflexões de Householder. O núcleo do estudo se concentrará nos algoritmos iterativos para a extração dos autovalores, partindo de métodos fundamentais como a Iteração da Potência (e sua variante inversa) e o Quociente de Rayleigh, e progredindo para abordagens mais robustas como a Iteração Simultânea e, principalmente, o Algoritmo QR, que será analisado em suas versões com e sem *shifts*. O algoritmo QR mostra-se uma peça central para a Álgebra Linear Numérica.

Por fim, apresenta-se uma comparação da eficiência dos métodos estudados. Os algoritmos descritos anteriormente serão avaliados com base em critérios como velocidade de convergência, custo computacional por iteração e precisão numérica, oferecendo uma análise prática sobre as vantagens e desvantagens de cada abordagem para diferentes tipos de problemas.

2 Fundamentos teóricos de autovalores

O estudo de autovalores e autovetores ocupa posição central na Álgebra Linear e na Análise Numérica. Esses conceitos fornecem não apenas uma descrição fundamental da estrutura de transformações lineares, mas também desempenham papel importante em aplicações. Como referência teórica neste capítulo, serão utilizados (GOLUB; LOAN, 2013), (STRANG, 2023), (HIGHAM, 2002) e (TREFETHEN; BAU, 1997).

Definição 1. *Dada uma matriz quadrada $A \in \mathbb{R}^{n\times n}$, diz-se que um escalar $\lambda \in \mathbb{C}$ é um* autovalor *de A se existe um vetor não nulo $x \in \mathbb{C}^n$ tal que*

$$Ax = \lambda x \tag{2.1}$$

O vetor x é chamado de autovetor *associado ao autovalor λ.*

O cálculo dos autovalores de A é feito através da equação (2.1) de forma que:

$$Ax = \lambda x \quad \Rightarrow \quad Ax - \lambda x = 0 \quad \Rightarrow \quad (A - \lambda I)x = 0$$

em que I é a matriz identidade de ordem n. O resultado anterior gera um sistema linear homogêneo, ou seja, sempre tem solução trivial $x = 0$ e terá uma solução não trivial apenas quando $(A - \lambda I)$ não for invertível, mas isso só ocorre quando $\det(A - \lambda I) = 0$.

Definição 2. *Uma matriz $A \in \mathbb{C}^{n\times n}$ é chamada* invertível *se existe uma matriz $B \in \mathbb{C}^{n\times n}$ tal que*

$$A\,B = B\,A = I_n,$$

onde I_n é a matriz identidade de ordem n.

Definição 3. *Uma matriz quadrada $A \in \mathbb{C}^{n\times n}$ é chamada* singular *se não for invertível, ou seja, se*

$$\det(A) = 0.$$

Definição 4. *Uma matriz $A \in \mathbb{C}^{n\times n}$ é dita* normal *se*

$$A^*A = AA^*,$$

em que $A^* = (\bar{a}_{ji})$ se $A = a_{ij}$.

Definição 5. *Uma matriz $A \in \mathbb{C}^{n\times n}$ é chamada* hermitiana *se*

$$A = A^*.$$

Definição 6. *Uma matriz $H \in \mathbb{C}^{n\times n}$ está em forma de* Hessenberg *(ou é chamada matriz de Hessenberg) se ela é quase triangular, isto é, no caso da forma superior de Hessenberg, se*

$$h_{i,j} = 0 \quad \textit{sempre que } i > j+1,$$

ou seja,

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1n} \\ h_{21} & h_{22} & h_{23} & \cdots & h_{2n} \\ 0 & h_{32} & h_{33} & \cdots & h_{3n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{n,n-1} & h_{nn} \end{bmatrix}.$$

Definição 7. *Para uma matriz $A = (a_{ij}) \in \mathbb{C}^{m\times n}$, a* norma de Frobenius *de A é definida por*

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}.$$

2.1 Polinômio característico

Definição 8. *Seja uma matriz quadrada $A \in \mathbb{C}^{n\times n}$. O* polinômio característico *é definido como*

$$p_A(\lambda) = \det(A - \lambda I) \tag{2.2}$$

de forma que o coeficiente do termo de grau n é 1.

Teorema 9. *λ é um autovalor de A se, e somente se, $p_A(\lambda) = 0$.*

Para cada autovalor λ, resolve-se $(A - \lambda I)x = 0$ para encontrar o autovetor correspondente.

Seja $A \in \mathbb{C}^{n\times n}$ com n autovalores $\lambda_1, \ldots, \lambda_n$ e autovetores linearmente independentes $x_1, \ldots, x_n$. Neste caso,

$$X = \left[\begin{array}{c|c|c} & & \\ x_1 & \cdots & x_n \\ & & \end{array}\right] \qquad \Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}.$$

Tem-se então

$$AX = X\Lambda. \tag{2.3}$$

Se X é invertível, obtém-se a *decomposição em autovalores*:

$$A = X\Lambda X^{-1}. \tag{2.4}$$

Definição 10. *Seja $A \in \mathbb{C}^{n\times n}$ e seja*

$$p_A(t) = \det(A - tI) = (t-\lambda_1)^{m_a(\lambda_1)}(t-\lambda_2)^{m_a(\lambda_2)}\cdots(t-\lambda_r)^{m_a(\lambda_r)}$$

a fatoração de p_A em raízes (contando multiplicidades), onde $\lambda_1,\ldots,\lambda_r$ são os autovalores distintos de A. A multiplicidade algébrica *$(m_a(\lambda_j))$ de um autovalor λ_j é o número de vezes que λ_j aparece como raiz de p_A.*

Um autovalor é dito *simples* quando sua multiplicidade algébrica for 1.

Definição 11. *Seja $A \in \mathbb{C}^{n\times n}$ e seja $\lambda \in \mathbb{C}$ um autovalor de A. O* autoespaço *associado a λ é dado por*

$$E_\lambda = \ker(A - \lambda I) = \{\, v \in \mathbb{C}^n \mid (A-\lambda I)v = 0 \,\}.$$

A multiplicidade geométrica *de λ, denotada por $m_g(\lambda)$, é definida como a dimensão de E_λ, isto é,*

$$m_g(\lambda) = \dim\ker(A - \lambda I). \tag{2.5}$$

Teorema 12. *Seja $A \in \mathbb{C}^{n\times n}$. Então A possui n autovalores, contados com multiplicidade algébrica. Em particular, se todas as raízes do polinômio característico $p_A(t)$ são simples, então A possui n autovalores distintos.*

2.2 Transformações de similaridade

Seja $X \in \mathbb{C}^{n\times n}$, então $A \mapsto X^{-1}AX$ é chamado de *transformação de similaridade*. Duas matrizes A e B são similares se existe uma transformação de similaridade relacionando uma à outra, ou seja, existe $X \in \mathbb{C}^{n\times n}$ não singular de forma que $B = X^{-1}AX$.

Teorema 13. *Se X é não singular, então A e $X^{-1}AX$ têm o mesmo polinômio característico, autovalores e multiplicidades algébricas e geométricas.*

Demonstração. Sabe-se que:

$$p_{X^{-1}AX}(z) = \det(zI - X^{-1}AX) = \det\big(X^{-1}(zI - A)X\big) = \det(X^{-1})\det(zI-A)\det(X).$$

Como $\det(X^{-1})\det(X) = 1$, tem-se

$$p_{X^{-1}AX}(z) = \det(zI - A) = p_A(z).$$

Como os polinômios característicos são iguais, segue que os autovetores são os mesmos. Além disso, como o polinômio é idêntico, a multiplicidade algébrica de cada raiz também coincide. Resta provar que as multiplicidades geométricas também coincidem.

Seja $E_\lambda(A) = \ker(A - \lambda I)$ o autoespaço de A associado a $\lambda \in \mathbb{C}$. Para todo $v \in E_\lambda(A)$, temos

$$Av = \lambda v.$$

Multiplicando à esquerda por X^{-1}, temos

$$X^{-1}Av = \lambda X^{-1}v.$$

Definindo $u = X^{-1}v$, então $v = Xu$. Assim,

$$(X^{-1}AX)u = \lambda u,$$

isto é, $u \in E_\lambda(X^{-1}AX)$. Assim, vale a inclusão

$$X^{-1}E_\lambda(A) \subseteq E_\lambda(X^{-1}AX).$$

O mesmo raciocínio no sentido inverso (multiplicando por X) mostra que

$$E_\lambda(X^{-1}AX) \subseteq X^{-1}E_\lambda(A).$$

Logo,

$$E_\lambda(X^{-1}AX) = X^{-1}E_\lambda(A).$$

Como X é não singular, a aplicação $v \mapsto X^{-1}v$ é bijetiva, e portanto preserva dimensões. Assim,

$$\dim E_\lambda(A) = \dim E_\lambda(X^{-1}AX).$$

Concluímos que as multiplicidades geométricas também coincidem. □

Teorema 14. *A multiplicidade algébrica de um autovalor é pelo menos tão grande quanto a sua multiplicidade geométrica.*

Demonstração. Denotemos por $E_\lambda(A) = \ker(A - \lambda I)$ o autoespaço de A associado a λ, e seja

$$n = \dim E_\lambda(A),$$

a multiplicidade geométrica de λ.

Escolha uma base ortonormal $\{v_1, \ldots, v_n\}$ de $E_\lambda(A)$ e construa a matriz

$$\widehat{V} = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} \in \mathbb{C}^{m \times n}.$$

Completaremos este conjunto de vetores com vetores ortonormais adicionais, de modo a obter uma base ortonormal de $\mathbb{C}^m$. Formando, assim, uma matriz unitária $V \in \mathbb{C}^{m \times m}$ que pode ser escrita como

$$V = \begin{bmatrix} \widehat{V} & W \end{bmatrix},$$

onde $W \in \mathbb{C}^{m\times(m-n)}$.

Consideremos a matriz semelhante

$$B = V^*AV.$$

Ao escrever B em blocos conformes à partição das colunas de V, obtemos

$$B = \begin{bmatrix} \widehat{V}^*A\widehat{V} & \widehat{V}^*AW \\ W^*A\widehat{V} & W^*AW \end{bmatrix}.$$

Como cada coluna de $\widehat{V}$ é autovetor associado a λ, temos $A\widehat{V} = \lambda\widehat{V}$. Assim:

$$\widehat{V}^*A\widehat{V} = \lambda I_n, \quad \text{e} \quad W^*A\widehat{V} = \lambda W^*\widehat{V} = 0,$$

pois W é ortogonal a $\widehat{V}$. Portanto, B assume a forma

$$B = \begin{bmatrix} \lambda I_n & C \\ 0 & D \end{bmatrix},$$

onde $C \in \mathbb{C}^{n\times(m-n)}$ e $D \in \mathbb{C}^{(m-n)\times(m-n)}$.

Agora, o polinômio característico de B pode ser calculado como

$$\det(zI - B) = \det(zI_n - \lambda I_n)\,\det(zI_{m-n} - D) = (z-\lambda)^n \det(zI_{m-n} - D).$$

Isto mostra que o fator $(z-\lambda)^n$ aparece no polinômio característico de B. Logo, a multiplicidade algébrica de λ como autovalor de B é pelo menos n.

Como A e B são semelhantes, eles têm o mesmo polinômio característico. Portanto, a multiplicidade algébrica de λ em A também é pelo menos n.

Assim, concluímos que $m_a(\lambda) \geq m_g(\lambda)$. □

Um autovalor cuja multiplicidade algébrica excede sua multiplicidade geométrica é um *autovalor defectivo*. Uma matriz que tem um ou mais autovalores deste tipo é chamada de *matriz defectiva*. Qualquer matriz diagonal é não defectiva.

2.3 Determinante e traço

O determinante de uma matriz $A \in \mathbb{C}^{m\times m}$ pode ser visto como o produto de seus autovalores (contados com multiplicidade algébrica), enquanto o traço de A, definido como a soma de seus elementos diagonais, coincide com a soma de seus autovalores. Essas igualdades fornecem uma ponte entre as operações elementares da Álgebra Linear e a teoria espectral.

Teorema 15. *Seja $A \in \mathbb{C}^{m\times m}$ com autovalores (contando multiplicidade algébrica) $\lambda_1, \dots, \lambda_m$. Então*

$$\det(A) = \prod_{j=1}^{m} \lambda_j \qquad e \qquad \operatorname{tr}(A) = \sum_{j=1}^{m} \lambda_j.$$

Demonstração. Considere o polinômio característico de A,

$$p_A(z) = \det(zI - A).$$

Pelo Teorema Fundamental da Álgebra, p_A fatora em $\mathbb{C}$ como

$$p_A(z) = \prod_{j=1}^{m} (z - \lambda_j). \tag{2.6}$$

Por outro lado, expandindo $\det(zI - A)$ em função dos coeficientes simétricos dos elementos de A, obtemos a forma canônica

$$p_A(z) = z^m - (\operatorname{tr} A)\, z^{m-1} + \cdots + (-1)^m \det(A). \tag{2.7}$$

Comparando as expressões (2.6) e (2.7) coeficiente a coeficiente, temos:

1. O coeficiente de z^{m-1} em (2.6) vale $-\sum_{j=1}^{m} \lambda_j$, enquanto em (2.7) vale $-\operatorname{tr}(A)$. Logo,

$$\operatorname{tr}(A) = \sum_{j=1}^{m} \lambda_j.$$

2. O termo constante em (2.6) é $\prod_{j=1}^{m}(-\lambda_j) = (-1)^m \prod_{j=1}^{m} \lambda_j$, enquanto em (2.7) é $(-1)^m \det(A)$. Assim,

$$\det(A) = \prod_{j=1}^{m} \lambda_j.$$

Essas identidades valem independentemente de A ser diagonalizável, pois decorrem exclusivamente da fatoração de p_A e da definição de traço e determinante via coeficientes do polinômio característico. □

2.4 Matrizes diagonalizáveis

As matrizes não defectivas são aquelas que satisfazem (2.4). Matrizes com essa propriedade são ditas *diagonalizáveis*. Em outras palavras, dizemos que uma matriz A é *diagonalizável* quando é semelhante a uma matriz diagonal. Ser diagonalizável é, portanto, equivalente a A possuir m autovetores linearmente independentes, o que ocorre precisamente quando A não é defectiva.

Teorema 16. *Seja $A \in \mathbb{C}^{m\times m}$. A matriz A é não defectiva se, e somente se, ela admite uma decomposição em autovalores da forma*

$$A = X\Lambda X^{-1},$$

onde X é não singular e Λ é diagonal.

Demonstração. ($\Rightarrow$) Suponha que A admita uma decomposição em autovalores, isto é,

$$A = X\Lambda X^{-1},$$

com X invertível e Λ diagonal. Pelo Teorema (13), matrizes semelhantes têm o mesmo polinômio característico, os mesmos autovalores e as mesmas multiplicidades.

Como Λ é diagonal, cada autovalor λ aparece na diagonal tantas vezes quanto sua multiplicidade algébrica, e os vetores canônicos formam uma base de autovetores para Λ. Portanto, Λ não é defectiva. Logo, A não é defectiva, pois a propriedade é preservada por similaridade.

($\Leftarrow$) Suponha agora que A seja não defectiva. Por definição, isso significa que, para cada autovalor λ de A, a sua multiplicidade geométrica é igual à sua multiplicidade algébrica: $m_g(\lambda) = m_a(\lambda)$. Como a soma das multiplicidades algébricas é igual a m, segue que existem exatamente m autovetores linearmente independentes de A.

Organizando esses m autovetores como colunas de uma matriz $X \in \mathbb{C}^{m\times m}$, temos que X é não singular. Além disso, pela definição de autovetor, obtemos $AX = X\Lambda$, onde Λ é uma matriz diagonal contendo os autovalores de A.

Multiplicando à direita por X^{-1}, concluímos que

$$A = X\Lambda X^{-1}.$$

Portanto, tem-se que A admite uma decomposição em autovalores se, e somente se, não é defectiva. □

2.5 Diagonalização unitária

Uma matriz A de dimensão $m \times m$ que possui m autovetores linearmente independentes e ortogonais é chamada de *unitariamente diagonalizável*, ou seja, existe uma matriz unitária Q tal que $A = Q\Lambda Q^*$.

Teorema 17. *Uma matriz hermitiana é unitariamente diagonalizável e seus autovalores são reais.*

Teorema 18. *Uma matriz é unitariamente diagonalizável se, e somente se, for normal, $A^*A = AA^*$.*

2.6 Fatoração de Schur

Embora nem toda matriz seja diagonalizável, é possível transformar qualquer matriz quadrada, via similaridade unitária, em uma matriz triangular superior. Essa decomposição é chamada de *fatoração de Schur* e afirma que para toda matriz $A \in \mathbb{C}^{m\times m}$, existe uma matriz unitária Q tal que

$$A = QTQ^* \tag{2.8}$$

onde Q é unitária e T é triangular superior cujos elementos diagonais são exatamente os autovalores de A.

Teorema 19. *Toda matriz quadrada $A \in \mathbb{C}^{m\times m}$ admite uma fatoração de Schur, isto é, existem uma matriz unitária Q e uma matriz triangular superior T tais que*

$$A = QTQ^*,$$

onde os elementos diagonais de T são exatamente os autovalores de A.

Demonstração. A demonstração será feita por indução em m, a dimensão da matriz A.

Caso base: Se $m = 1$, o resultado é trivial: basta tomar $Q = [1]$ e $T = A$.

Passo indutivo: Suponha que o resultado seja válido para todas as matrizes de dimensão menor que m, e considere $A \in \mathbb{C}^{m\times m}$ com $m \geq 2$.

Como A é complexa, ela possui ao menos um autovalor λ e um autovetor não nulo x. Normalizamos x de modo que $\|x\| = 1$, e o escolhemos como a primeira coluna de uma matriz unitária U. Com isso, pode-se verificar que

$$U^*AU = \begin{bmatrix} \lambda & B \\ 0 & C \end{bmatrix},$$

onde $C \in \mathbb{C}^{(m-1)\times(m-1)}$ e B é o bloco $1 \times (m-1)$ formado pelos elementos restantes da primeira linha de U^*AU. .

Pela hipótese de indução, existe uma fatoração de Schur para C, isto é, existem uma matriz unitária V e uma triangular superior T tais que

$$C = VTV^*.$$

Agora definimos a matriz unitária em blocos

$$W = \begin{bmatrix} 1 & 0 \\ 0 & V \end{bmatrix}.$$

Então,

$$W^*(U^*AU)W = \begin{bmatrix} \lambda & BV \\ 0 & T \end{bmatrix},$$

que é triangular superior, com λ e os autovalores de C na diagonal.

Por fim, definimos

$$Q = UW,$$

que é unitária, e obtemos

$$A = Q\widetilde{T}Q^*,$$

onde $\widetilde{T} = W^*(U^*AU)W$ é triangular superior com os autovalores de A na diagonal.

Assim, pelo princípio de indução, toda matriz quadrada A admite uma fatoração de Schur. $\square$

As fatorações que revelam autovalores — diagonalização, diagonalização unitária e fatoração de Schur — constituem a base teórica para o estudo de métodos numéricos de cálculo espectral. Entre essas, a fatoração de Schur destaca-se por sua aplicabilidade geral e por preservar a estabilidade numérica, uma vez que envolve apenas transformações unitárias. Assim, todo algoritmo robusto para o cálculo de autovalores parte, direta ou indiretamente, da busca por uma forma quase triangular obtida via transformações unitárias sucessivas.

Contudo, aplicar diretamente o processo de Schur a uma matriz arbitrária é computacionalmente ineficiente. Para contornar essa limitação, introduz-se uma etapa intermediária de redução: antes de triangularizar A, reduz-se a matriz a uma forma estruturalmente mais simples, mas que preserva seus autovalores. Essa forma é conhecida como forma de Hessenberg.

3 Fase 1: triangularização de matrizes

O cálculo numérico de autovalores e autovetores é uma das tarefas centrais da Álgebra Linear Numérica. No entanto, aplicar diretamente os métodos teóricos sobre a matriz original $A \in \mathbb{R}^{n \times n}$ é, em geral, computacionalmente ineficiente e numericamente instável. Por essa razão, os algoritmos modernos de autovalores são estruturados em duas fases principais:

1) uma fase que transforma a matriz A por similaridade, preservando o espectro, em uma matriz reduzida (na forma Hessenberg, ou tridiagonal no caso simétrico);

2) uma fase iterativa que, também por similaridade, extrai da matriz reduzida os autovalores desejados.

Esta primeira etapa, denominada *triangularização*, é responsável por converter A em uma forma que facilite o processamento subsequente — uma matriz de Hessenberg (no caso geral) ou tridiagonal (para matrizes simétricas).

A transformação é conduzida por meio de operações ortogonais:

$$Q^T A Q = H,$$

em que Q é uma matriz ortogonal ($Q^T Q = I$) e H é a matriz reduzida. Essas transformações preservam os autovalores e a estabilidade numérica, pois mantêm inalteradas as normas e o condicionamento do problema.

A técnica que será utilizada para efetuar tal redução será as transformações de Householder, que permitem eliminar sistematicamente os elementos indesejados de A sem comprometer sua estrutura espectral.

3.1 Reflexões de Householder

O método das reflexões de Householder é uma técnica para a construção da decomposição QR de matrizes (dos Santos, 2025). Esse método utiliza uma sequência de transformações unitárias que introduzem zeros abaixo da diagonal principal de uma matriz, preservando sua norma e ortogonalidade (TREFETHEN; BAU, 1997).

A ideia central consiste em aplicar, sucessivamente, refletores de Householder Q_k projetados de modo que o produto

$$Q_n Q_{n-1} \cdots Q_1 A$$

resulte em uma matriz triangular superior R, enquanto o produto acumulado das transformações unitárias gera uma matriz ortogonal Q, de forma que

$$A = QR.$$

Cada refletor Q_k atua sobre as linhas $k, \ldots, m$ da matriz, eliminando os elementos abaixo da diagonal na k-ésima coluna. Para isso, constrói-se uma matriz unitária elementar F, definida por

$$F = I - 2\frac{vv^*}{v^*v},$$

onde v é um vetor ortogonal ao hiperplano de reflexão, dado por

$$v = \|x\|e_1 - x,$$

sendo x o vetor coluna atual. Essa transformação reflete o vetor x em torno do hiperplano ortogonal a v, produzindo um novo vetor

$$Fx = \|x\|e_1,$$

no qual as componentes abaixo da diagonal são anuladas.

Geometricamente, o refletor de Householder realiza uma reflexão especular: os pontos de um lado do hiperplano são mapeados em seus correspondentes do lado oposto, preservando distâncias e ângulos entre vetores. Essa propriedade garante a estabilidade numérica e a preservação da norma.

Em termos computacionais, o método é eficiente, possui bom comportamento quanto ao condicionamento e aos erros de arredondamento. Por essas razões, as reflexões de Householder constituem a base de diversos algoritmos modernos de álgebra linear numérica, como os métodos QR iterativos para autovalores e a decomposição SVD.

3.2 Redução à forma de Hessenberg

O procedimento de redução de uma matriz quadrada geral $A \in \mathbb{R}^{n\times n}$ à sua forma de Hessenberg é realizado por meio de reflexões de Householder. Essa forma reduzida constitui o ponto de partida para os métodos iterativos de cálculo de autovalores, como o algoritmo QR, pois mantém o espectro de A e reduz significativamente o custo computacional das iterações subsequentes.

A estratégia na Fase 1 é introduzir zeros de maneira gradual, atuando apenas sobre os elementos necessários da matriz. O objetivo não é triangularizar completamente A, mas transformá-la em uma matriz de Hessenberg H com zeros abaixo da primeira subdiagonal, preservando a similaridade ortogonal:

$$Q^T A Q = H, \quad \text{com } Q^T Q = I.$$

Dessa forma, H é semelhante a A e, portanto, possui o mesmo conjunto de autovalores.

O refletor de Householder Q_i é construído de forma a introduzir zeros abaixo da subdiagonal da coluna i. Seja x o vetor formado pelos elementos da coluna i de A a partir da linha $i+1$, isto é,

$$x = \begin{bmatrix} a_{i+1,i} \\ a_{i+2,i} \\ \vdots \\ a_{n,i} \end{bmatrix} \in \mathbb{R}^{n-i}.$$

Deseja-se encontrar uma transformação ortogonal que leve x em um múltiplo do primeiro vetor canônico e_1, ou seja,

$$\widehat{Q}_i x = \pm \|x\|_2 e_1.$$

Para isso, define-se o vetor de Householder

$$v_i = \frac{x \pm \|x\|_2 e_1}{\|x \pm \|x\|_2 e_1\|_2},$$

e o refletor correspondente

$$\widehat{Q}_i = I - 2 v_i v_i^T,$$

onde $\widehat{Q}_i \in \mathbb{R}^{(n-i)\times(n-i)}$ atua apenas sobre as linhas e colunas inferiores da matriz. Assim, o refletor completo Q_i tem a estrutura em blocos

$$Q_i = \begin{bmatrix} I_i & 0 \\ 0 & \widehat{Q}_i \end{bmatrix},$$

preservando as i primeiras linhas e colunas de A.

A escolha do sinal em v_i é feita de modo a evitar cancelamento numérico, garantindo maior estabilidade na transformação. Multiplicando A à esquerda e à direita por Q_i, os elementos abaixo da posição $(i+1, i)$ são anulados, preservando a similaridade ortogonal e aproximando a matriz de sua forma de Hessenberg.

No primeiro passo, seleciona-se um refletor de Householder Q_1 que mantém a primeira linha inalterada e atua apenas sobre as linhas $2, 3, \ldots, m$, introduzindo zeros nas posições $(3, 1), (4, 1), \ldots, (m, 1)$. Multiplicando A à esquerda por Q_1^T, obtém-se:

$$A \xrightarrow{Q_1^T} Q_1^T A = \begin{bmatrix} \times & \times & \times & \cdots \\ \times & \times & \times & \cdots \\ 0 & \times & \times & \cdots \\ 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

Em seguida, multiplicando à direita por Q_1, preserva-se a primeira coluna e os zeros introduzidos, obtendo:

$$Q_1^T A Q_1 = \begin{bmatrix} \times & \times & \times & \cdots \\ \times & \times & \times & \cdots \\ 0 & \times & \times & \cdots \\ 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

O processo é então repetido para as colunas subsequentes. O segundo refletor Q_2 atua sobre as linhas e colunas $2, 3, \ldots, m$, zerando os elementos abaixo da posição $(3, 2)$, sem alterar a estrutura obtida nas colunas anteriores:

$$Q_2^T Q_1^T A Q_1 Q_2 = \begin{bmatrix} \times & \times & \times & \times & \cdots \\ \times & \times & \times & \times & \cdots \\ 0 & \times & \times & \times & \cdots \\ 0 & 0 & \times & \times & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

Após $m - 2$ iterações, obtém-se:

$$Q_{m-2}^T \cdots Q_2^T Q_1^T A Q_1 Q_2 \cdots Q_{m-2} = H,$$

onde H é a matriz de Hessenberg desejada. Definindo

$$Q = Q_1 Q_2 \cdots Q_{m-2},$$

temos a relação final

$$Q^T A Q = H.$$

A Fase 1 do processo de cálculo de autovalores tem como objetivo fundamental transformar a matriz original $A \in \mathbb{R}^{n \times n}$ em uma forma mais simples, porém espectralmente equivalente. Por meio das transformações ortogonais de Householder, reduz-se A à sua forma de Hessenberg (ou tridiagonal, no caso simétrico), mantendo os autovalores inalterados e garantindo estabilidade numérica.

Essa etapa, embora não forneça ainda os autovalores de A, cria as condições ideais para a fase seguinte. A estrutura quase triangular da matriz reduzida H permite a aplicação eficiente de métodos iterativos, como o algoritmo QR, que exploram a forma de Hessenberg para reduzir o custo computacional.

4 Fase 2: Iteração QR e Extração de Autovalores

A Fase 2 corresponde ao núcleo do processo de cálculo numérico de autovalores. Após a redução da matriz original $A \in \mathbb{R}^{n\times n}$ à forma de Hessenberg na Fase 1, o problema espectral é reformulado de maneira a permitir um procedimento iterativo estável e computacionalmente eficiente. A ideia central consiste em aplicar, de forma repetida, fatorações QR sucessivas à matriz reduzida, de modo que, à medida que as iterações avançam, ela tenda a uma forma quase triangular (ou diagonal, no caso simétrico), cujos elementos diagonais aproximam os autovalores de A.

Matematicamente, a iteração QR baseia-se no seguinte esquema iterativo, dado A_k, calcula-se:

$$A_k = Q_k R_k \quad \text{e depois} \quad A_{k+1} = R_k Q_k,$$

em que Q_k é ortogonal e R_k é triangular superior. Como cada passo envolve apenas transformações ortogonais, a similaridade é preservada:

$$A_{k+1} = Q_k^T A_k Q_k,$$

o que implica que todas as matrizes A_k compartilham o mesmo conjunto de autovalores de A. Com o decorrer das iterações, os elementos fora da diagonal tendem a zero e A_k converge para uma matriz triangular superior, expondo diretamente os autovalores na diagonal.

Para os métodos apresentados nesta fase, consideraremos o caso particular de matrizes hermitianas (reais e simétricas), pois ele conduz a propriedades espectrais e computacionais mais favoráveis. Assim, assumimos que

$$A = A^* \in \mathbb{R}^{m\times m}, \qquad x \in \mathbb{R}^m, \qquad \|x\| = \|x\|_2 = \sqrt{x^* x}.$$

Nessas condições, A possui apenas autovalores reais e um conjunto completo de autovetores ortogonais, normalizados de modo que $\|q_j\| = 1$. Denotaremos por

$$\lambda_1, \lambda_2, \ldots, \lambda_m \quad \text{os autovalores reais, e por} \quad q_1, q_2, \ldots, q_m \quad \text{os autovetores ortonormais.}$$

Devido à Fase 1 e ao fato de A ser hermitiana, a matriz de entrada para esta etapa já se encontra na forma tridiagonal, pois ela é simétrica e Hessenberg. Isso reduz o custo computacional de cada iteração e simplifica a análise teórica dos algoritmos.

A Fase 2, portanto, completa o processo iniciado na triangularização: enquanto a primeira etapa preparou a matriz para o cálculo espectral, esta segunda fase realiza efetivamente a extração dos autovalores e autovetores.

Os próximos métodos apresentados são ferramentas úteis para a construção do algoritmo QR que será descrito. Para isso, serão utilizadas as fontes (PANJU, 2011) e (TREFETHEN; BAU, 1997).

4.1 Quociente de Rayleigh

Os métodos iterativos funcionam refinando repetidamente as estimativas de autovalores de uma matriz simétrica, usando uma função chamada quociente de Rayleigh. Dado um vetor não nulo $x \in \mathbb{R}^n$ e uma matriz real e simétrica $A \in \mathbb{R}^{n\times n}$, o *quociente de Rayleigh* é definido por:

$$r(x) = \frac{x^T A x}{x^T x}. \tag{4.1}$$

Esse quociente fornece uma estimativa do autovalor de A associado ao vetor x. Note que se x é um autovetor, então $r(x) = \lambda$ é o autovalor correspondente.

4.2 Iteração da potência

Suponha que o conjunto q_i de autovetores unitários de A forma uma base de $\mathbb{R}^n$ e tem autovalores reais correspondentes λ_i tais que

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|. \tag{4.2}$$

Seja $v^{(0)}$ uma aproximação de um autovetor de A, com $\|v^{(0)}\| = 1$. Escrevendo $v^{(0)}$ como uma combinação linear dos autovetores ortonormais q_i:

$$v^{(0)} = a_1 q_1 + a_2 q_2 + \cdots + a_m q_m. \tag{4.3}$$

Então,

$$Av^{(0)} = a_1\lambda_1 q_1 + a_2\lambda_2 q_2 + \cdots + a_m\lambda_m q_m \tag{4.4}$$

e assim,

$$A^k v^{(0)} = a_1\lambda_1^k q_1 + a_2\lambda_2^k q_2 + \cdots + a_m\lambda_m^k q_m = \lambda_1^k \left(a_1 q_1 + a_2 \left(\frac{\lambda_2^k}{\lambda_1^k}\right) q_2 + \cdots + a_m \left(\frac{\lambda_m^k}{\lambda_1^k}\right) q_m \right).$$

Como os autovalores são assumidos como reais, distintos e ordenados por magnitude decrescente, segue-se que para todo $i = 2, \ldots, n$,

$$\lim_{k\to\infty} \left(\frac{\lambda_i}{\lambda_1}\right)^k = 0.$$

Então, à medida que k aumenta, $A^k v^{(0)}$ se aproxima de $c_1\lambda_1^k q_1$, e assim para grandes valores de k,

$$q_1 \approx \frac{A^k v^{(0)}}{\|A^k v^{(0)}\|}.$$

Por si só, a iteração de potência é de uso limitado por várias razões. Primeiro, ela só pode encontrar o autovetor correspondente ao maior autovalor. Segundo, a convergência é linear, reduzindo o erro apenas por um fator constante com valor aproximado a $\left|\frac{\lambda_2}{\lambda_1}\right|$ a cada iteração.

Após a apresentação teórica do método de iteração da potência, apresentamos a seguir uma aplicação numérica com o objetivo de observar empiricamente o seu comportamento. Para isso, construiremos uma matriz simétrica $A \in \mathbb{R}^{n\times n}$ cujos autovalores são previamente especificados, o que nos permite controlar a razão espectral $|\lambda_2/\lambda_1|$ e, consequentemente, a taxa de convergência do método.

A matriz é gerada a partir da decomposição

$$A = QDQ^T,$$

na qual $D = \operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ e Q é uma matriz ortogonal obtida via fatoração QR de uma matriz aleatória. Essa construção garante que A seja real, simétrica e possua exatamente os autovalores desejados. O código correspondente a essa implementação encontra-se no Apêndice (A) desta monografia.

Ao aplicar o método da iteração da potência sobre a matriz construída, obtemos aproximações sucessivas para o autovalor dominante λ_1 e para o autovetor associado. O critério de parada utilizado baseia-se no resíduo espectral

$$\|Av^{(k)} - \lambda^{(k)}v^{(k)}\|_2,$$

que mede a precisão da aproximação obtida a cada iteração.

Para o conjunto de autovalores escolhido,

$$\{5.7,\ 5.6,\ 3,\ 2\},$$

a razão espectral

$$\rho = \left|\frac{\lambda_2}{\lambda_1}\right| \approx 0.982$$

indica que a convergência do método será relativamente lenta.

Tabela 1 – Tabela de convergência da iteração da potência.

Iteração	**Erro** $\lvert\lambda^{(k)} - \lambda^{(\text{final})}\rvert$	**Razão de erros**
1	1.19×10^{0}	0
11	2.53×10^{-2}	9.73×10^{-1}
21	1.89×10^{-2}	9.70×10^{-1}
31	1.37×10^{-2}	9.67×10^{-1}
41	9.70×10^{-3}	9.64×10^{-1}
51	6.62×10^{-3}	9.61×10^{-1}
61	4.33×10^{-3}	9.56×10^{-1}
71	2.65×10^{-3}	9.48×10^{-1}
81	1.44×10^{-3}	9.32×10^{-1}
91	5.68×10^{-4}	8.84×10^{-1}
98	1.11×10^{-4}	6.55×10^{-1}

A Tabela (1) resultante apresenta a evolução do erro $\lvert\lambda^{(k)} - \lambda_1\rvert$ e a razão entre erros sucessivos, evidenciando a lenta convergência linear característica do método.

4.3 Método da potência inversa com *shift*

O método da potência inversa é uma modificação do método da potência, projetada para localizar o autovalor de menor magnitude e o autovetor correspondente de uma matriz. Enquanto o método da potência aplica repetidamente a matriz A a um vetor inicial, fazendo prevalecer o autovetor associado ao maior autovalor em módulo, o método da potência inversa baseia-se na aplicação iterativa da matriz inversa A^{-1}, que inverte o papel dos autovalores.

Considere uma matriz $A \in \mathbb{R}^{n\times n}$ com autovalores reais e distintos

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Os autovalores de A^{-1} são $1/\lambda_1, 1/\lambda_2, \ldots, 1/\lambda_n$. Assim, o maior autovalor em módulo de A^{-1} corresponde ao menor autovalor em módulo de A. Aplicando o método da potência a A^{-1}, obtemos uma aproximação para o autovetor associado ao menor autovalor de A.

Introduzindo um deslocamento (*shift*) nos autovalores antes de inverter $(A - \mu I)^{-1}$ permite fazer com que o autovetor q_j associado a λ_j de A passe a ser associado ao autovalor $\frac{1}{\lambda_j - \mu}$ levando ao método da potência inversa com *shift*.

A partir de agora nos referiremos ao método da potência inversa com *shift* apenas como método da potência inversa.

Na prática, o algoritmo não requer a inversão explícita da matriz, o que seria computacionalmente custoso e numericamente instável. Em vez disso, a cada iteração, resolve-se um sistema linear:

$$(A - \mu I)y^{(k)} = x^{(k-1)},$$

seguido de normalização:

$$x^{(k)} = \frac{y^{(k)}}{\|y^{(k)}\|}.$$

A sequência $\{x^{(k)}\}$ converge para o autovetor associado ao autovalor de A mais próximo de μ. Uma estimativa desse autovalor pode ser obtida, novamente, pelo coeficiente de Rayleigh:

$$\lambda^{(k)} = \rho(x^{(k)}) = \frac{(x^{(k)})^T A x^{(k)}}{(x^{(k)})^T x^{(k)}}.$$

O método da potência inversa é particularmente eficaz quando o autovalor de interesse está próximo de μ, permitindo localizar autovalores interiores.

Para ilustrar a eficiência do método da potência inversa, apresentamos a seguir um exemplo computacional completo, cujo código-fonte encontra-se no apêndice (B). Nesse experimento, consideramos uma matriz simétrica construída de modo a possuir autovalores próximos entre si, situação na qual o método da potência tradicional apresenta convergência lenta devido à razão entre os dois maiores autovalores em módulo ser próxima de 1.

A matriz utilizada no experimento foi construída de forma a possuir autovalores predeterminados:

$$\{5.7,\ 5.6,\ 3,\ 2\}.$$

Foi gerado uma matriz ortogonal Q por meio da fatoração QR de uma matriz aleatória. Em seguida, definimos

$$A = Q\,\mathrm{diag}(5.7,\ 5.6,\ 3,\ 2)\,Q^T,$$

garantindo assim que A seja simétrica e tenha exatamente os autovalores especificados.

Neste experimento, empregamos um valor fixo de *shift*

$$\mu = 5.58,$$

o qual determina o autovalor para o qual o método deve convergir: o autovalor de A que está mais próximo de μ. O autovalor λ alvo foi obtido ao final das iterações por meio do quociente de Rayleigh $\lambda^{(k)} = v^{(k)T} A v^{(k)}$.

Para estudar a taxa de convergência, registramos o histórico dos valores $\lambda^{(k)}$ ao longo das iterações. Definimos o erro na iteração k como

$$E_k = \lambda^{(k+1)} - \lambda,$$

isto é, a diferença entre a aproximação no passo $k+1$ e o autovalor final λ. A razão de erros exibida na tabela é calculada por

$$\frac{E_{k+1}}{E_k} = \frac{\lambda^{(k+2)} - \lambda}{\lambda^{(k+1)} - \lambda}.$$

Ao aplicar a iteração inversa com um *shift* adequadamente escolhido, observou-se uma convergência significativamente mais rápida para o autovalor de interesse, bem como um erro residual consideravelmente menor quando comparado aos resultados obtidos pelo método da potência. Além disso, o coeficiente de Rayleigh forneceu estimativas progressivamente mais precisas ao longo das iterações, evidenciando a robustez e a vantagem numérica da técnica em cenários desafiadores.

Na Tabela (2) encontram-se os valores aproximados do autovalor ao longo das iterações, bem como a razão dos erros, permitindo avaliar empiricamente a taxa de convergência do método.

Tabela 2 – Tabela de convergência potência inversa

Iteração	$\boldsymbol{\lambda^{(k)} - \lambda}$	**Razão de erros**
1	3.759383×10^{-3}	3.7879×10^{-2}
2	1.424005×10^{-4}	2.7824×10^{-2}
3	3.962108×10^{-6}	2.7779×10^{-2}
4	1.100628×10^{-7}	2.7778×10^{-2}
5	3.057305×10^{-9}	2.7778×10^{-2}
6	8.492496×10^{-11}	2.7778×10^{-2}
7	2.359809×10^{-12}	2.8227×10^{-2}
8	6.661338×10^{-14}	4.0000×10^{-2}
9	2.664535×10^{-15}	6.6667×10^{-1}
10	1.776357×10^{-15}	0

4.4 Método do Quociente de Rayleigh

O método de iteração do quociente de Rayleigh pode ser interpretado como uma extensão do método da potência inversa com *shift*. Para isso, abandona-se a restrição de que o valor de *shift* permaneça constante ao longo das iterações. Cada iteração do método de iteração inversa com *shift* retorna um autovetor aproximado, dado uma estimativa de um autovalor. O quociente de Rayleigh, por outro lado, produz um autovalor aproximado quando recebe um autovetor estimado. Ao combinar essas duas operações, obtemos uma nova variação do algoritmo de *shift* inverso, onde o valor de *shift* é recomputado durante cada iteração para se tornar o quociente de Rayleigh da estimativa atual do autovetor. Esse método funciona da seguinte forma:

Dado um vetor inicial normalizado $x^{(0)} \neq 0$ e uma matriz simétrica $A \in \mathbb{R}^{n \times n}$, inicia-se o processo calculando o quociente de Rayleigh

$$\mu^{(0)} = \rho(x^{(0)}) = \frac{(x^{(0)})^T A x^{(0)}}{(x^{(0)})^T x^{(0)}}.$$

Em seguida, resolve-se o sistema linear deslocado

$$(A - \mu^{(0)} I) y^{(0)} = x^{(0)},$$

obtendo-se um novo vetor, que é normalizado para produzir a próxima aproximação

$$x^{(1)} = \frac{y^{(0)}}{\|y^{(0)}\|}.$$

Essas etapas são repetidas iterativamente, sendo o *shift* μ_k recalculado a cada passo a partir do vetor mais recente, ou seja,

$$\mu^{(k)} = \frac{(x^{(k)})^T A x^{(k)}}{(x^{(k)})^T x^{(k)}}.$$

Dessa forma, o valor de μ_k acompanha dinamicamente o autovalor para o qual a sequência $\{x^{(k)}\}$ está convergindo.

Para ilustrar o comportamento prático da iteração do quociente de Rayleigh, aplicamos o método à matriz simétrica construída a partir de autovalores prescritos

$$\{5.7,\ 5.6,\ 3,\ 2\},$$

conforme o código apresentado no Apêndice (C). A escolha dessa matriz permite observar a rápida convergência do algoritmo, mesmo quando os autovalores são relativamente próximos entre si.

Nesse experimento numérico, partimos de um vetor inicial aleatório e executamos o método até que o critério de parada baseado no erro residual fosse satisfeito. A cada iteração, o valor do *shift* é atualizado dinamicamente a partir do quociente de Rayleigh associado ao autovetor aproximado corrente, conduzindo o processo a um refinamento progressivamente mais preciso da estimativa do autovalor dominante.

Para avaliar a taxa de convergência do método, calculamos um estimador para o expoente de convergência p, o qual indica o quão rapidamente o erro diminui a cada iteração. Seja $E_k = |\lambda_k - \lambda|$ o erro relativo ao autovalor aproximado na iteração k. Podemos estimar p por meio da relação assintótica

$$E_{k+1} \approx C E_k^p,$$

o que implica

$$p \approx \frac{\log\left(\frac{E_{k+1}}{E_k}\right)}{\log\left(\frac{E_k}{E_{k-1}}\right)}.$$

Assim, para cada iteração $k \geq 3$, o valor de p foi obtido utilizando os erros consecutivos das iterações anteriores. Valores não definidos (NaN) surgem quando o erro é nulo ou quando a razão entre erros não pode ser computada numericamente.

A Tabela (3) apresenta os *shifts* μ_k que representam as estimativas sucessivas do autovalor, bem como os erros, as razões entre erros consecutivos e o expoente de

convergência estimado. Como esperado da teoria, observa-se uma convergência superlinear, aproximando-se do comportamento cúbico $p \approx 3$ característico da iteração do quociente de Rayleigh para matrizes simétricas. Além disso, nota-se que o método atinge elevado grau de precisão em poucas iterações, superando de maneira significativa os resultados obtidos anteriormente com a iteração da potência e a iteração inversa com *shift* fixo.

Tabela 3 – Tabela de convergência do quociente de Rayleigh

Iteração	μ_k	$\lambda_k - \lambda$	**Razão de erros**	p
1	2.81×10^0	8.05×10^{-1}	4.29×10^{-1}	NaN
2	2.35×10^0	3.46×10^{-1}	3.03×10^{-1}	NaN
3	2.10×10^0	1.05×10^{-1}	1.51×10^{-2}	3.51×10^0
4	2.00×10^0	1.59×10^{-3}	2.53×10^{-6}	3.08×10^0
5	2.00×10^0	4.01×10^{-9}	5.54×10^{-8}	1.30×10^0
6	2.00×10^0	-2.22×10^{-16}	NaN	NaN

4.5 Iteração simultânea

Os métodos de iteração discutidos anteriormente, como o método da potência, são projetados para calcular apenas um autovalor (e seu autovetor associado) de cada vez. Para encontrar múltiplos autovalores distintos, seria necessário reaplicar o método diversas vezes, o que é ineficiente.

O método da iteração simultânea surge como uma alternativa capaz de aproximar, sob certas condições, múltiplos autovetores de uma matriz simultaneamente.

Para descrever esse método, toma-se como base o método de iteração de potência. Seja A uma matriz real, simétrica e de posto completo. No método da potência, um vetor inicial $v^{(0)}$ pode ser escrito como combinação linear de autovetores $\{q_i\}$ de A conforme a equação (4.3).

A ideia inicial da iteração simultânea é generalizar esse processo. Em vez de um único vetor, selecionamos uma base de n vetores linearmente independentes, $\{v_1^{(0)} \dots v_n^{(0)}\}$ que formam as colunas da matriz $V^{(0)}$:

$$V^{(0)} = \left[\begin{array}{c|c|c|c} & & & \\ v_1^{(0)} & v_2^{(0)} & \cdots & v_n^{(0)} \\ & & & \end{array}\right]$$

Aplicar a iteração de potência a todos esses vetores simultaneamente equivaleria a calcular:

$$V^{(k)} = A^k V^{(0)} = \left[\begin{array}{c|c|c|c} \\ v_1^{(k)} & v_2^{(k)} & \cdots & v_n^{(k)} \\ \\ \end{array} \right].$$

Contudo, essa abordagem apresenta um problema fundamental. Como o método da potência converge para o autovetor associado ao autovalor de maior magnitude (λ_1), todas as colunas de $V^{(k)}$ tenderiam a convergir para a mesma direção, a do autovetor dominante q_1. O resultado seria uma matriz $V^{(k)}$ cujas colunas se tornariam quase linearmente dependentes (praticamente paralelas), falhando em encontrar os demais autovetores.

A solução para esse colapso de vetores é forçar que as colunas da matriz de aproximação permaneçam ortogonais em cada iteração. Para isso, será utilizada a decomposição QR.

O algoritmo é modificado da seguinte forma:

1. Comece com uma matriz $\hat{Q}^{(0)}$, obtida através do fator $\hat{Q}$ de uma decomposição QR reduzida de uma $V^{(0)}$ aleatória, cujas colunas formam uma base ortonormal para $\mathbb{R}^n$.
2. Para $k = 1, 2, ...$, repita os seguintes passos:
   a) **Iteração Potência:** Calcule a matriz $Z = A\hat{Q}^{(k-1)}$. Este passo aplica a transformação A ao subespaço ortonormal da iteração anterior.
   b) **Decomposição QR:** Calcule a decomposição QR da matriz Z, obtendo $Z = \hat{Q}^{(k)}\hat{R}^{(k)}$.
3. A matriz $\hat{Q}^{(k)}$ contém a nova base ortonormal de estimativas dos autovetores.

Ao forçar a ortogonalidade em cada etapa por meio da decomposição QR, garantimos que os vetores-coluna de $\hat{Q}^{(k)}$ permaneçam linearmente independentes. Sob condições bem gerais, é possível mostrar que as colunas de $\hat{Q}^{(k)}$ convergirão para os k primeiros autovetores de A. A taxa de convergência, por sua vez, depende da razão entre o $k+1$-ésimo e o k-ésimo autovalor $\left|\frac{\lambda_{k+1}}{\lambda_k}\right|$.

4.6 QR sem shifts

Aqui explicaremos o algoritmo QR para o cálculo de autovalores, o algoritmo consiste na sequência de passos:

Dado $A^{(0)} = A$, repetimos:

1. Faça a fatoração QR da matriz atual:

$$A^{(k)} = Q^{(k)}R^{(k)}, \qquad Q^{(k)} \text{ ortogonal, } R_k \text{ triangular superior.}$$

2. Defina a próxima matriz da iteração por

$$A^{(k+1)} = R^{(k)}Q^{(k)}.$$

Observe que $A^{(k+1)} = (Q^{(k)})^T A^{(k)} Q^{(k)}$, isto é, cada passo produz uma matriz semelhante à anterior, preservando seu espectro. A repetição deste processo tende a aproximar uma matriz triangular superior, cujas entradas diagonais aproximam os autovalores procurados.

No que se segue, mostraremos que o algoritmo QR descrito acima é equivalente a iteração simultânea aplicada a um conjunto completo de vetores iniciais $n = m$.

Como as matrizes $\hat{Q}^{(k)}$ agora são quadradas, a decomposição QR será completa. Os acentos de $\hat{R}^{(k)}$ e $\hat{Q}^{(k)}$ serão substituídos por $R^{(k)}$ e $\underline{Q}^{(k)}$, respectivamente. A barra embaixo do $\underline{Q}$ serve para indicar a matriz ortogonal da iteração simultânea, enquanto o Q indica a da iteração QR.

A sequência de fórmulas a seguir define a iteração simultânea com $Q^{(0)} = I$ e na qual a matriz $A^{(k)}$ é $m \times m$.

$$\underline{Q}^{(0)} = I, \tag{4.5}$$

$$Z = A\underline{Q}^{(k-1)}, \tag{4.6}$$

$$Z = \underline{Q}^{(k)}R^{(k)} \tag{4.7}$$

$$A^{(k)} = (\underline{Q}^{(k)})^T A\underline{Q}^{(k)} \tag{4.8}$$

Agora, as três fórmulas iniciais definem o algoritmo QR puro.

$$A^{(0)} = A, \tag{4.9}$$

$$A^{(k-1)} = Q^{(k)}R^{(k)}, \tag{4.10}$$

$$A^{(k)} = R^{(k)}Q^{(k)}, \tag{4.11}$$

Além disso, em ambos os algoritmos, pode-se definir uma matriz $\underline{R}^{(k)}, m \times m$, como:

$$\underline{R}^{(k)} = R^{(k)}R^{(k-1)} \dots R^{(1)}. \tag{4.12}$$

e uma sequência $\underline{Q}^{(k)}$ como:

$$\underline{Q}^{(k)} = Q^{(1)}Q^{(2)} \cdots Q^{(k)}. \tag{4.13}$$

O teorema a seguir demonstra a equivalência entre o método de iteração QR sem shift e a iteração simultânea.

Teorema 20. *Os processos (4.5)–(4.8) e (4.9)–(4.13) geram sequências idênticas das matrizes $\underline{R}^{(k)}$, $\underline{Q}^{(k)}$, e $A^{(k)}$, especificamente, aquelas definidas pela fatoração QR da k-ésima potência de A,*

$$A^k = \underline{Q}^{(k)} \underline{R}^{(k)}, \tag{4.14}$$

juntamente com

$$A^{(k)} = (\underline{Q}^{(k)})^T A \underline{Q}^{(k)}. \tag{4.15}$$

Demonstração. Procedemos por indução em k.

Para $k = 0$, é claro que $A^{(0)}$, $\underline{Q}^{(0)}$, e $\underline{R}^{(0)}$ são os mesmos para ambos os algoritmos, o método da iteração simultânea e o método QR, e esses valores satisfazem (4.14) e (4.15).

Suponha agora que os valores dessas matrizes são os mesmos para ambos os algoritmos em alguma iteração $k-1$, e que eles satisfazem as duas propriedades (4.14) e (4.15) nesta iteração.

No método da iteração simultânea, usando a hipótese que (4.14) vale para $k-1$ e (4.6), temos

$$\begin{aligned} A^k &= AA^{k-1} \\ &= A(\underline{Q}^{(k-1)} \underline{R}^{(k-1)}) \\ &= (A\underline{Q}^{(k-1)}) \underline{R}^{(k-1)} \\ &= \underline{Q}^{(k)} R^{(k)} \underline{R}^{(k-1)} \\ &= \underline{Q}^{(k)} \underline{R}^{(k)} \end{aligned}$$

e assim a propriedade (4.14) é satisfeita na k-ésima iteração; a propriedade (4.15) é satisfeita diretamente pela definição do algoritmo de iteração simultânea.

No método QR, usando hipótese, temos

$$\begin{aligned} A^k &= AA^{k-1} \\ &= A(Q^{(1)}Q^{(2)} \cdots Q^{(k-1)} R^{(k-1)} \cdots R^{(1)}) \\ &= (Q^{(1)}R^{(1)})(Q^{(1)}Q^{(2)} \cdots Q^{(k-1)} R^{(k-1)} \cdots R^{(1)}), \end{aligned}$$

como $A^{(1)} = R^{(1)}Q^{(1)}$ e $A^{(1)} = Q^{(2)}R^{(2)}$,

$$\begin{aligned} A^k &= Q^{(1)}(R^{(1)}Q^{(1)})Q^{(2)} \cdots Q^{(k-1)} R^{(k-1)} \cdots R^{(1)} \\ &= Q^{(1)}(Q^{(2)}R^{(2)})Q^{(2)} \cdots Q^{(k-1)} R^{(k-1)} \cdots R^{(1)} \\ &= Q^{(1)}Q^{(2)}(R^{(2)}Q^{(2)}) \cdots Q^{(k-1)} R^{(k-1)} \cdots R^{(1)} \end{aligned}$$

e $A^{(3)} = R^{(2)}Q^{(2)}$, etc...

$$\begin{aligned} A^k &= Q^{(1)}Q^{(2)}Q^{(3)} \cdots Q^{(k-1)}(Q^{(k)}R^{(k)})R^{(k-1)} \cdots R^{(1)} \\ &= \underline{Q}^{(k)} \underline{R}^{(k)} \end{aligned}$$

o que prova que a propriedade (4.14) vale para a k-ésima iteração do método QR. Nós também temos

$$\begin{aligned} A^{(k)} &= R^{(k)}Q^{(k)} \\ &= \left((Q^{(k)})^T Q^{(k)}\right) R^{(k)}Q^{(k)} \\ &= (Q^{(k)})^T \left(Q^{(k)}R^{(k)}\right) Q^{(k)} \\ &= (Q^{(k)})^T A^{(k-1)} Q^{(k)} \end{aligned}$$

que pela hipótese de indução é igual a:

$$\begin{aligned} A^{(k)} &= (Q^{(k)})^T \left((\underline{Q}^{(k-1)})^T A \underline{Q}^{(k-1)}\right) Q^{(k)} \\ &= (\underline{Q}^{(k)})^T A \underline{Q}^{(k)} \end{aligned}$$

o que prova que a propriedade (4.15) vale para a k-ésima iteração do método QR também.

Por hipótese, os valores de $A^{(k-1)}$, $\underline{Q}^{(k-1)}$, e $\underline{R}^{(k-1)}$ eram os mesmos para ambos os algoritmos. Como ambos os algoritmos satisfazem (4.14) na k-ésima iteração, temos $A^k = \underline{Q}^{(k)}\underline{R}^{(k)}$ para ambos os algoritmos, e como a decomposição QR é única, segue que $\underline{Q}^{(k)}$ e $\underline{R}^{(k)}$ são também os mesmos para ambos os algoritmos. Ambos os algoritmos também satisfazem (4.15) na k-ésima iteração, o que significa que a matriz $A^{(k)} = (\underline{Q}^{(k)})^T A \underline{Q}^{(k)}$ é também a mesma para ambos os algoritmos na k-ésima iteração.

Portanto, ambos os algoritmos produzem os mesmos valores para as matrizes $A^{(k)}$, $\underline{Q}^{(k)}$, e $\underline{R}^{(k)}$ que satisfazem as relações (4.14) e (4.15). Por indução, isso vale para todo k, provando o teorema. □

Com a equivalência entre o método QR e a iteração simultânea estabelecida no teorema (20), podemos compreender porque o algoritmo QR sem shift converge. As relações (4.14) e (4.15) são as chaves para essa compreensão.

1. A equação $A^k = \underline{Q}^{(k)}\underline{R}^{(k)}$ demonstra que o algoritmo QR está, implicitamente, construindo as bases ortonormais para as potências sucessivas A^k. Este é o mesmo mecanismo da iteração da potência, explicando porque as colunas de $\underline{Q}^{(k)}$ convergem para uma base de autovetores de A.

2. A equação $A^{(k)} = (\underline{Q}^{(k)})^T A \underline{Q}^{(k)}$ revela que as matrizes $A^{(k)}$ são projeções de A sobre subespaços que aproximam os auto-espaços de A. Mais importante, os elementos diagonais de $A^{(k)}$ são, segundo a definição, os Quocientes de Rayleigh de A correspondentes às colunas de $\underline{Q}^{(k)}$, ou seja, são aproximações dos autovalores.

A conclusão é que, à medida que as colunas de $\underline{Q}^{(k)}$ convergem para os autovetores, os elementos diagonais de $A^{(k)}$ convergem para os autovalores correspondentes.

Embora tenhamos estabelecido que o algoritmo QR sem *shift* converge, sua taxa de convergência na prática é, muitas vezes, lenta demais para ser eficiente. A velocidade de convergência depende diretamente da razão entre autovalores consecutivos ($|\lambda_{j+1}|/|\lambda_j|$). Se os autovalores estiverem próximos em magnitude, o método pode levar um número proibitivo de iterações.

Para superar essa deficiência, e acelerar drasticamente a convergência, uma modificação fundamental é introduzida: a técnica de *shift*.

Por fim, para ilustrar numericamente o comportamento do método QR sem *shift*, no Apêndice (D) está um exemplo computacional construído especificamente com esse propósito. A matriz utilizada no experimento é simétrica e foi gerada de modo a possuir exatamente os mesmos autovalores empregados nos exemplos anteriores, ou seja,

$$\{5.7,\ 5.6,\ 3,\ 2\}.$$

Após sucessivas iterações do método, observa-se que os elementos fora da diagonal principal decaem gradualmente. Entretanto, como discutido teoricamente, a convergência do método QR puro é tipicamente linear, e sua taxa está intimamente relacionada com a razão espectral

$$\left|\frac{\lambda_2}{\lambda_1}\right|,$$

o que pode ser observado na tabela de erros apresentada ao longo das iterações. Como consequência, o método pode demandar um número elevado de iterações quando os autovalores estão relativamente próximos em magnitude, comportamento confirmado no experimento computacional.

Tabela 4 – Convergência do método QR sem *shifts*.

Iteração	$\lambda_k - \lambda$	Razão de erros
1	1.9247×10^{-1}	1.4119
2	2.7176×10^{-1}	2.6598
3	7.2281×10^{-1}	1.0242
4	7.4031×10^{-1}	5.2832×10^{-1}
5	3.9112×10^{-1}	3.5700×10^{-1}
10	6.6539×10^{-4}	8.3729×10^{-2}
20	2.6890×10^{-4}	9.7082×10^{-1}
30	1.9842×10^{-4}	9.6924×10^{-1}
40	1.4435×10^{-4}	9.6809×10^{-1}
50	1.0394×10^{-4}	9.6726×10^{-1}
60	7.4288×10^{-5}	9.6666×10^{-1}
70	5.2815×10^{-5}	9.6624×10^{-1}
80	3.7407×10^{-5}	9.6594×10^{-1}
90	2.6423×10^{-5}	9.6573×10^{-1}
99	1.9293×10^{-5}	9.6559×10^{-1}

Esse exemplo numérico evidencia a limitações práticas do método, pois o erro decai lentamente como $C(0.96)^k$ onde k é o número de iterações e C constante. Isso motiva aprimoramentos posteriores, como a introdução de *shifts*.

4.7 Método QR com Shift

O método QR com *shift* é uma das variantes mais eficientes do algoritmo QR para o cálculo de autovalores de matrizes reais e simétricas. A principal ideia consiste em substituir temporariamente a matriz A pela matriz deslocada $A - \mu I$, onde μ é uma estimativa para um autovalor de A. Esse *shift* altera o espectro da matriz, acelerando a convergência do algoritmo para determinados autovalores.

A ideia é aplicar a fatoração QR não a $A^{(k)}$, mas sim à matriz $A^{(k)} - \mu^{(k)} I$.

Dado $A^{(0)} = A$, para cada iteração realizamos:

1. Escolha um shift $\mu^{(k)}$.

2. Forme a decomposição QR da matriz deslocada:

$$A^{(k)} - \mu^{(k)} I = Q^{(k)} R^{(k)}.$$

3. Reconstrua a nova matriz aplicando o shift de volta:

$$A^{(k+1)} = R^{(k)} Q^{(k)} + \mu^{(k)} I.$$

A introdução de shifts aproxima implicitamente o comportamento do método QR ao da iteração inversa com *shift*, conhecida por fornecer convergência mais rápida quando um bom palpite para μ está disponível. Nesse contexto, a escolha adequada do shift desempenha papel fundamental. Um dos critérios mais utilizados é o quociente de Rayleigh, definido por

$$\mu^{(k)} = \frac{x^{(k)T} A x^{(k)}}{x^{(k)T} x^{(k)}},$$

em que $x^{(k)}$ representa o autovetor aproximado na iteração k. Quando aplicado de forma iterativa, esse shift induz convergência cúbica na aproximação de autovalores simples, tornando o método substancialmente mais eficiente em comparação à versão sem *shifts*.

Apesar de seu excelente desempenho para a maioria das matrizes, o shift baseado no quociente de Rayleigh pode apresentar dificuldades em casos de simetrias espectrais específicas, causando ciclos ou falta de progresso. Para contornar tais situações, é comum empregar o shift de Wilkinson.

4.7.1 Shift de Wilkinson

O *shift* de Wilkinson é uma técnica utilizada no método QR com *shift* para o cálculo eficiente de autovalores de matrizes reais e simétricas. Seu objetivo principal é acelerar a convergência das iterações ao selecionar, de forma criteriosa, um *shift* escalar μ que aproxima um dos autovalores da matriz, reduzindo assim a quantidade de iterações necessárias para a chamada "deflação" que detalharemos a seguir.

Considere uma matriz simétrica tridiagonal $A \in \mathbb{R}^{n\times n}$ resultante da fase 1 de redução de Hessenberg ou tridiagonalização. Seja

$$A = \begin{pmatrix} * & * & & & \\ * & * & * & & \\ & \ddots & \ddots & \ddots & \\ & & * & * & * \\ & & & \mathbf{*} & * \end{pmatrix},$$

onde apenas os elementos da diagonal principal e diagonais secundárias são não nulos. Em tal situação, a deflação ocorre quando o elemento subdiagonal $a_{n,n-1}$ (em negrito) torna-se suficientemente pequeno, isto é,

$$|a_{n,n-1}| < \varepsilon,$$

para algum ε determinado. Neste caso, o autovalor correspondente pode ser extraído, reduzindo o problema ao sub-bloco superior esquerdo de dimensão $(n-1) \times (n-1)$.

O shift de Wilkinson constrói μ a partir dos autovalores da submatriz 2×2

$$B = \begin{pmatrix} a_{n-1,n-1} & a_{n-1,n} \\ a_{n,n-1} & a_{n,n} \end{pmatrix}.$$

Denotando tais autovalores por λ_1 e λ_2, o shift é definido como aquele autovalor de B mais próximo do elemento $a_{n,n}$ da diagonal principal de A. De forma equivalente, o *shift* pode ser computado resolvendo o polinômio característico de B:

$$\lambda = \frac{a_{n-1,n-1} + a_{n,n}}{2} \pm \sqrt{\left(\frac{a_{n-1,n-1} - a_{n,n}}{2}\right)^2 + a_{n,n-1}^2}.$$

A seleção do sinal é importante: escolhe-se o autovalor que minimiza $|\lambda - a_{n,n}|$. Tal critério tende a direcionar a convergência para o autovalor de menor módulo relativo ao bloco inferior, induzindo o elemento $a_{n,n-1}$ a decair rapidamente para zero.

Além do ganho de velocidade, o shift de Wilkinson apresenta excelente estabilidade numérica. Por envolver apenas operações locais sobre um bloco 2×2, ele minimiza a propagação de erros de arredondamento.

Durante as iterações QR, a adoção do shift de Wilkinson produz frequentemente deflações naturais: após algumas iterações, o elemento $a_{n,n-1}$ torna-se numericamente nulo,

permitindo a extração imediata de um autovalor sem necessidade de etapas adicionais. Esse processo reduz o tamanho da matriz ativa, diminuindo progressivamente o custo computacional.

Nas seções a seguir mostraremos o desempenho do método QR com *shift* de Wilkinson em dois casos com graus distintos de dificuldade. Um exemplo ilustrando o passo a passo do método para uma matriz 5×5 está descrito no capítulo 5.

4.8 Exemplo "fácil"

Após a descrição teórica do Método QR com shifts e da motivação matemática por trás do shift de Wilkinson, apresentamos aqui uma implementação prática em `Python` aplicada ao caso simétrico. O objetivo é ilustrar o comportamento numérico do método em uma situação simples, com uma matriz cuja estrutura espectral favorece a convergência.

A matriz utilizada neste exemplo é uma matriz simétrica $A \in \mathbb{R}^{25\times 25}$, construída de modo controlado a partir de autovalores conhecidos. Esses autovalores são definidos artificialmente por

$$\lambda_k = \frac{(-1)^{k+1}}{2^{k+1}}, \quad k = 1, 2, \ldots, 25,$$

de modo que decaem exponencialmente e alternam de sinal. Tal escolha produz um espectro bem distribuído, no qual a razão entre autovalores consecutivos é $\frac{1}{2}$, evitando dificuldades numéricas causadas por autovalores próximos um dos outros.

A partir desse conjunto de autovalores, forma-se a matriz diagonal

$$D = \operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_{25}).$$

Em seguida, constrói-se uma matriz ortogonal aleatória Q por meio da fatoração QR de uma matriz aleatória para garantir colunas ortonormais. Por fim, a matriz de teste é obtida por similaridade:

$$A = QDQ^T.$$

Com isso, garante-se que A é real, simétrica e possui exatamente o espectro definido por $\{\lambda_k\}$. Além disso, a escolha aleatória de Q impede que a matriz apresente qualquer estrutura artificial que favoreça o algoritmo.

O código completo do apêndice (F) está dividido em duas fases clássicas do método QR:

1. **Redução à forma tridiagonal**, realizada por reflexões de Householder.
2. **Iteração QR com shift de Wilkinson**, aplicada sobre a matriz tridiagonal resultante.

A primeira etapa é executada pela função `householder_tridiagonal`, que transforma a matriz simétrica A em uma matriz tridiagonal T semelhante a A, preservando seus autovalores.

Na segunda etapa, a função `practical_qr_symmetric` realiza sucessivas fatorações QR na matriz tridiagonal $T^{(k)}$ atualizada:

$$T^{(k)} - \mu^{(k)} I = Q^{(k)} R^{(k)},$$

atualizando-a segundo

$$T^{(k+1)} = R^{(k)} Q^{(k)} + \mu^{(k)} I.$$

O parâmetro $\mu^{(k)}$ é o *shift* de Wilkinson, calculado a partir do bloco 2×2 inferior de $T^{(k)}$. Esse shift acelera a deflação do elemento $T^{(k)}_{m-1,m-2}$, fazendo com que ele convirja para zero de forma quadrática. Quando tal elemento se torna menor que uma tolerância pré-definida, ocorre a *deflação*, ou seja, o tamanho efetivo m da matriz é reduzido em uma unidade, isolando um autovalor convergido.

A convergência do método é monitorada por meio da norma de Frobenius dos elementos fora da diagonal da matriz tridiagonal em cada iteração. Essa norma fornece uma medida quantitativa do quão próxima a matriz $T^{(k)}$ está de ser diagonal, e é definida como:

$$\|T^{(k)} - \operatorname{diag}(T^{(k)})\|_F = \left(\sum_{i \neq j} |t_{ij}^{(k)}|^2 \right)^{1/2},$$

onde $t_{ij}^{(k)}$ denota o elemento (i, j) da matriz $T^{(k)}$ na k-ésima iteração.

Na implementação, o cálculo é realizado diretamente pela diferença entre a matriz tridiagonal corrente e sua diagonal principal, conforme:

$$\texttt{erro_offdiag} = \|T^{(k)} - \operatorname{diag}(T^{(k)})\|_F.$$

Quando esse valor se torna suficientemente pequeno, conclui-se que todos os elementos fora da diagonal se aproximaram de zero, indicando que a matriz $T^{(k)}$ está praticamente diagonalizada e que os autovalores de A foram obtidos com alta precisão numérica.

O comportamento da norma de Frobenius fora da diagonal ao longo das iterações fornece uma visão do processo de convergência do método QR com shifts. À medida que o algoritmo progride, as transformações ortogonais aplicadas reduzem gradualmente os elementos fora da diagonal de $T^{(k)}$, o que faz com que o valor de

$$\|T^{(k)} - \operatorname{diag}(T^{(k)})\|_F$$

diminua de forma sistemática.

Esse decaimento reflete a aproximação progressiva de $T^{(k)}$ por uma matriz diagonal, cujas entradas representam os autovalores de A. Em particular, a taxa de redução dessa

norma está diretamente relacionada à eficiência do shift de Wilkinson: quanto mais rápido o valor tende a zero, mais eficaz é o processo de deflação e isolamento dos autovalores.

A representação gráfica da Figura (1) dessa norma em escala logarítmica evidencia a natureza exponencial da convergência. O gráfico foi obtido automaticamente durante a execução do algoritmo do apêndice (F).

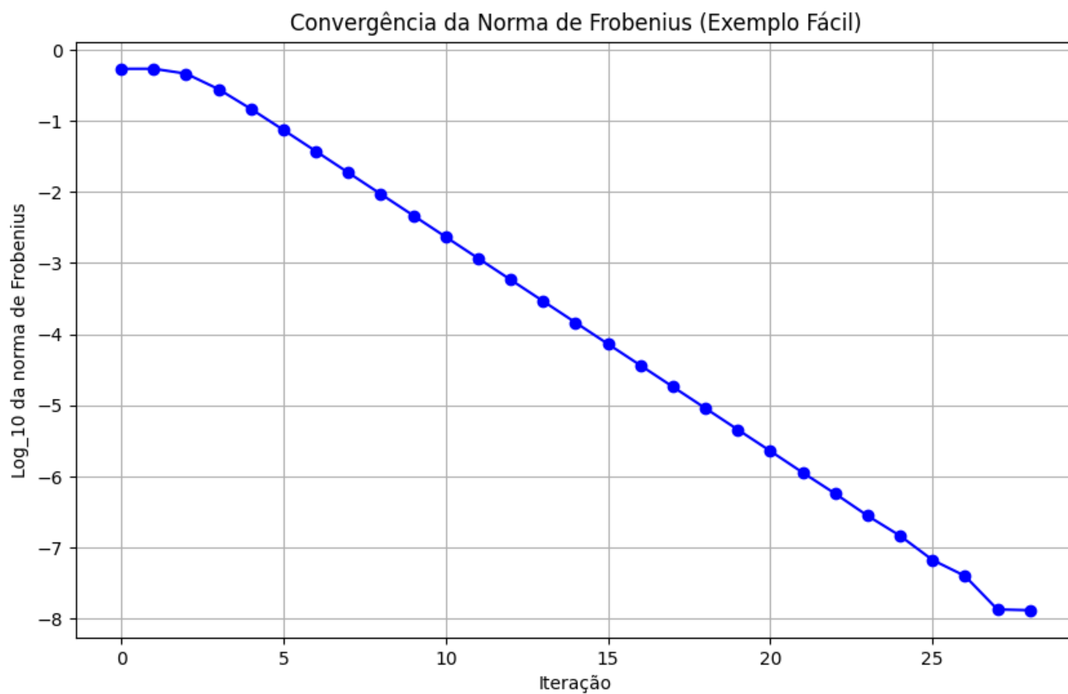

Figura 1 – Convergência da norma de Frobenius fora da diagonal ao longo das iterações do método QR com shift de Wilkinson.

A Tabela (5) mostra a sequência de convergência obtida automaticamente durante a execução do algoritmo do apêndice (F), evidenciando a deflação progressiva da matriz.

Iteração	μ (shift)	$\lvert\lambda_k - \lambda\rvert$	Valor de m
1	2.271×10^{-07}	9.543×10^{-09}	25
2	2.384×10^{-07}	1.445×10^{-12}	25
3	2.384×10^{-07}	5.689×10^{-24}	25
4	5.820×10^{-08}	5.415×10^{-10}	24
5	5.960×10^{-08}	1.267×10^{-15}	24
6	5.960×10^{-08}	3.609×10^{-32}	24
7	-2.980×10^{-08}	4.180×10^{-13}	23
8	-2.980×10^{-08}	9.571×10^{-29}	23
9	-1.192×10^{-07}	2.308×10^{-20}	22
10	-4.768×10^{-07}	1.076×10^{-18}	21
11	9.537×10^{-07}	1.044×10^{-22}	20
12	-1.907×10^{-06}	4.259×10^{-24}	19
13	3.815×10^{-06}	1.104×10^{-22}	18
14	-7.629×10^{-06}	7.905×10^{-25}	17
15	1.526×10^{-05}	1.958×10^{-26}	16
16	-3.052×10^{-05}	1.344×10^{-25}	15
17	6.104×10^{-05}	1.585×10^{-25}	14
18	-1.221×10^{-04}	7.538×10^{-27}	13
19	2.441×10^{-04}	1.535×10^{-26}	12
20	-4.883×10^{-04}	2.098×10^{-26}	11
21	9.766×10^{-04}	5.225×10^{-25}	10
22	-1.953×10^{-03}	3.823×10^{-26}	9
23	3.906×10^{-03}	6.725×10^{-27}	8
24	-7.812×10^{-03}	1.715×10^{-26}	7
25	1.563×10^{-02}	4.547×10^{-26}	6
26	-3.125×10^{-02}	4.767×10^{-26}	5
27	6.250×10^{-02}	3.166×10^{-26}	4
28	-1.250×10^{-01}	8.047×10^{-27}	3
29	2.500×10^{-01}	5.753×10^{-26}	2

Tabela 5 – Convergência do método QR com shift de Wilkinson para o exemplo fácil.

4.9 Exemplo "difícil"

Neste exemplo, analisamos o comportamento do método QR com shifts aplicado a uma matriz simétrica especialmente construída para representar um caso de convergência lenta. O objetivo é observar como a escolha dos autovalores influencia o desempenho do algoritmo e o processo de deflação.

Inicialmente, o algoritmo do apêndice (G) gera-se uma matriz ortogonal $Q \in \mathbb{R}^{n\times n}$ por meio da fatoração QR de uma matriz aleatória para garantir colunas ortonormais. A seguir, define-se uma matriz diagonal

$$D = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n),$$

onde os autovalores são dados por

$$\lambda_k = \cos\left(\frac{(k+1)\pi}{m}\right).$$

Com isso, constrói-se a matriz simétrica

$$A = QDQ^T.$$

Essa escolha torna o exemplo "difícil" porque os autovalores gerados pela função cosseno tornam-se muito próximos de 1 ou -1, dificultando a separação dos autovalores durante as iterações do método QR.

A função `householder_tridiagonal` aplica reflexões de Householder sobre A, eliminando os elementos abaixo da primeira subdiagonal, e assim reduzindo A a uma forma tridiagonal T semelhante a ela. Essa etapa constitui a Fase 1 do algoritmo.

Na Fase 2, a função `practical_qr_symmetric` realiza iterações do tipo

$$T^{(k+1)} = R^{(k)}Q^{(k)} + \mu^{(k)}I,$$

onde $(Q^{(k)}, R^{(k)}) = \mathrm{qr}(T^{(k)} - \mu^{(k)}I)$ e $\mu^{(k)}$ é o *shift* de Wilkinson. A cada passo, o algoritmo verifica a condição de deflação:

$$|T^{(k)}_{m-1,m-2}| \leq \mathrm{tol}\left(|T^{(k)}_{m-1,m-1}| + |T^{(k)}_{m-2,m-2}|\right).$$

Quando a condição é satisfeita, o elemento fora da diagonal é anulado e o tamanho efetivo m do bloco ativo é reduzido, representando a convergência de um autovalor.

Durante as iterações, o algoritmo também avalia o comportamento da norma de Frobenius considerando apenas os elementos fora da diagonal principal. Essa métrica mede o quanto a matriz atual $T^{(k)}$ ainda difere de uma matriz diagonal, e é calculada por:

$$\|T^{(k)} - \mathrm{diag}(T^{(k)})\|_F.$$

Valores pequenos dessa norma indicam que $T^{(k)}$ está se aproximando de uma matriz diagonal, isto é, que os autovalores estão sendo isolados com sucesso.

A Figura (2) ilustra a evolução da Norma de Frobenius Fora da Diagonal da matriz tridiagonal T ao longo das iterações do algoritmo QR prático com o *shift* de Wilkinson.

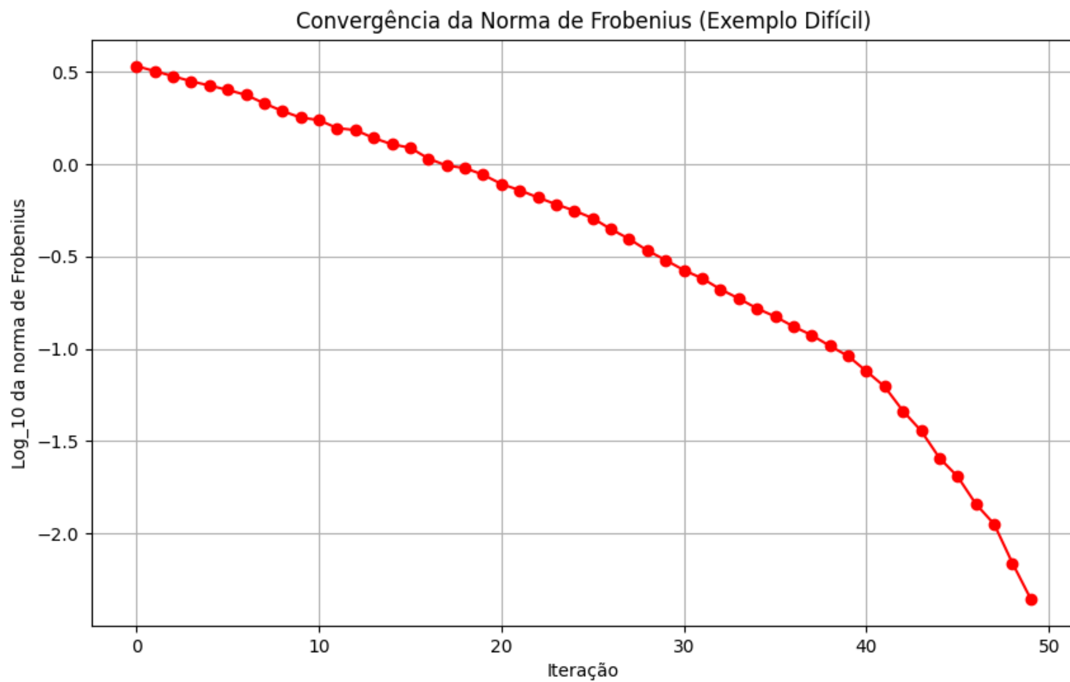

Figura 2 – Convergência da norma de Frobenius fora da diagonal ao longo das iterações (Exemplo Difícil).

A Tabela (6) apresenta, para cada iteração, o valor do *shift* $\mu^{(k)}$, o erro $|T^{(k)}_{m-1,m-2}|$ e o valor corrente de m, que diminui a cada deflação detectada.

Iteração	μ (shift)	$\lvert\lambda_k - \lambda\rvert$	Valor de m
1	-8.985×10^{-1}	3.558×10^{-2}	25
2	-9.234×10^{-1}	2.843×10^{-3}	25
3	-9.298×10^{-1}	5.640×10^{-7}	25
4	-9.298×10^{-1}	6.229×10^{-18}	25
5	-8.760×10^{-1}	5.892×10^{-5}	24
6	-8.763×10^{-1}	1.544×10^{-12}	24
7	-9.692×10^{-1}	1.252×10^{-4}	23
8	-9.686×10^{-1}	1.649×10^{-9}	23
9	-9.686×10^{-1}	6.869×10^{-26}	23
10	-9.922×10^{-1}	4.084×10^{-6}	22
11	-9.921×10^{-1}	1.984×10^{-12}	22
12	-1.000×10^{0}	5.041×10^{-10}	21
13	-1.000×10^{0}	3.502×10^{-27}	21
14	-8.090×10^{-1}	1.412×10^{-9}	20
15	-8.090×10^{-1}	4.393×10^{-25}	20
16	-7.290×10^{-1}	1.099×10^{-10}	19
17	-7.290×10^{-1}	1.543×10^{-28}	19
18	-6.374×10^{-1}	3.845×10^{-7}	18
19	-6.374×10^{-1}	8.564×10^{-21}	18
20	-5.358×10^{-1}	3.735×10^{-8}	17
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Iteração	μ (shift)	$\lvert\lambda_k - \lambda\rvert$	Valor de m
$\vdots$	$\vdots$	$\vdots$	$\vdots$
21	-5.358×10^{-1}	2.811×10^{-23}	17
22	-4.258×10^{-1}	4.752×10^{-7}	16
23	-4.258×10^{-1}	5.532×10^{-20}	16
24	-3.090×10^{-1}	2.601×10^{-7}	15
25	-3.090×10^{-1}	6.864×10^{-20}	15
26	-1.874×10^{-1}	1.659×10^{-6}	14
27	-1.874×10^{-1}	4.384×10^{-21}	14
28	-6.279×10^{-2}	5.722×10^{-12}	13
29	-6.279×10^{-2}	1.187×10^{-32}	13
30	6.279×10^{-2}	1.919×10^{-8}	12
31	6.279×10^{-2}	3.072×10^{-24}	12
32	1.874×10^{-1}	2.223×10^{-6}	11
33	1.874×10^{-1}	3.702×10^{-17}	11
34	3.091×10^{-1}	1.895×10^{-5}	10
35	3.090×10^{-1}	2.507×10^{-15}	10
36	4.258×10^{-1}	2.532×10^{-8}	9
37	4.258×10^{-1}	9.221×10^{-25}	9
38	5.358×10^{-1}	1.099×10^{-9}	8
39	5.358×10^{-1}	1.282×10^{-25}	8
40	6.374×10^{-1}	9.611×10^{-9}	7
41	6.374×10^{-1}	8.027×10^{-25}	7
42	7.290×10^{-1}	1.655×10^{-7}	6
43	7.290×10^{-1}	1.130×10^{-20}	6
44	8.090×10^{-1}	2.124×10^{-6}	5
45	8.090×10^{-1}	1.390×10^{-17}	5
46	8.763×10^{-1}	3.871×10^{-7}	4
47	8.763×10^{-1}	1.086×10^{-19}	4
48	9.298×10^{-1}	3.634×10^{-7}	3
49	9.298×10^{-1}	2.999×10^{-19}	3
50	9.686×10^{-1}	2.856×10^{-18}	2

Tabela 6 – Tabela de convergência do método QR com shift de Wilkinson (Exemplo Difícil).

A gráfico e a tabela deste exemplo foram extraídos do apêndice (G).

O gráfico a seguir ilustra como a razão entre autovalores afeta a velocidade de convergência do método QR com *shift* de Wilkinson.

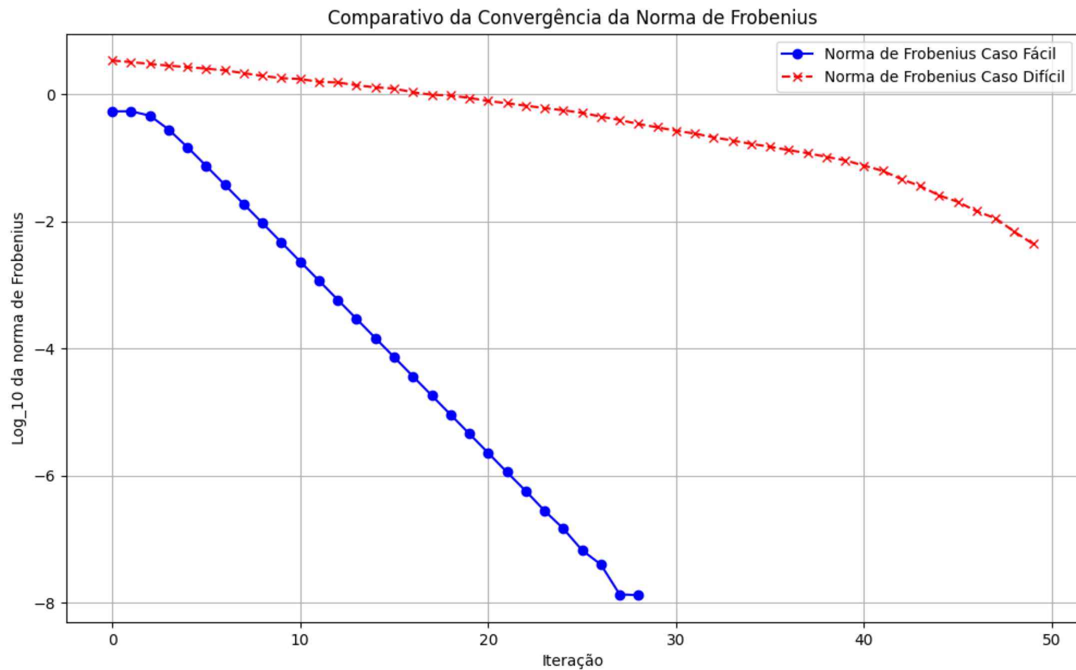

Figura 3 – Comparação da convergência da norma de Frobenius fora da diagonal ao longo das iterações.

No caso fácil (curva azul), os autovalores satisfazem

$$\left|\frac{\lambda_{k+1}}{\lambda_k}\right| = \frac{1}{2},$$

o que produz uma separação espectral significativa. Isso favorece o método, resultando em uma queda praticamente linear em escala logarítmica.

Por outro lado, no caso difícil (curva vermelha), os autovalores são dados por

$$\lambda_k = \cos\left(\frac{(k+1)\pi}{m}\right),$$

de modo que as razões entre autovalores são próximas de 1, indicando forte aglomeração espectral. Nessa situação, o *shift* de Wilkinson perde eficiência, e a convergência passa a ser mais lenta e irregular.

5 Aplicação Fase 1 e 2

Neste capítulo apresentamos um exemplo numérico completo que ilustra as duas fases do método desenvolvido: (i) a redução de uma matriz simétrica ao formato de Hessenberg utilizando reflexões de Householder e (ii) a etapa iterativa baseada no método QR com shift de Wilkinson e deflação. O objetivo é mostrar como o algoritmo converge para os autovalores. Este exemplo foi criado em *Python* cujo código utilizado encontra-se no apêndice (H).

A matriz inicial A foi construída a partir de uma base ortonormal aleatória, tomando

$$A = QDQ^\top,$$

onde $D = \text{diag}(5.7,\ 5.6,\ 3,\ 2,\ -1.99)$ contém os autovalores pré-definidos. Assim, a matriz resultante é simétrica por construção.

A matriz inicial A é:

$$\text{A} = \begin{bmatrix} 3.350102{\cdot}10^{0} & -4.191634{\cdot}10^{-1} & 1.557682{\cdot}10^{0} & 1.824064{\cdot}10^{0} & 9.461629{\cdot}10^{-1} \\ -4.191634{\cdot}10^{-1} & 2.596605{\cdot}10^{0} & 4.539367{\cdot}10^{-2} & 1.513938{\cdot}10^{0} & 5.924768{\cdot}10^{-1} \\ 1.557682{\cdot}10^{0} & 4.539367{\cdot}10^{-2} & 4.188715{\cdot}10^{0} & 5.599265{\cdot}10^{-2} & -9.310342{\cdot}10^{-1} \\ 1.824064{\cdot}10^{0} & 1.513938{\cdot}10^{0} & 5.599265{\cdot}10^{-2} & -7.022195{\cdot}10^{-1} & 4.146295{\cdot}10^{-1} \\ 9.461629{\cdot}10^{-1} & 5.924768{\cdot}10^{-1} & -9.310342{\cdot}10^{-1} & 4.146295{\cdot}10^{-1} & 4.876797{\cdot}10^{0} \end{bmatrix}.$$

5.1 Fase 1 — Redução à Forma de Hessenberg

Aplicamos as reflexões de Householder de forma sucessiva para anular os elementos abaixo da primeira subdiagonal. A implementação utilizada realizou três iterações até atingir a forma desejada. A matriz Hessenberg obtida foi:

$$\text{H} = \begin{bmatrix} 3.350102{\cdot}10^{0} & 2.612375{\cdot}10^{0} & 0 & 0 & 0 \\ 2.612375{\cdot}10^{0} & 1.290898{\cdot}10^{0} & -2.489047{\cdot}10^{0} & 0 & 0 \\ 0 & -2.489047{\cdot}10^{0} & 1.144490{\cdot}10^{0} & 4.569804{\cdot}10^{-1} & 0 \\ 0 & 0 & 4.569804{\cdot}10^{-1} & 3.155039{\cdot}10^{0} & -8.269709{\cdot}10^{-1} \\ 0 & 0 & 0 & -8.269709{\cdot}10^{-1} & 5.369472{\cdot}10^{0} \end{bmatrix}.$$

5.2 Fase 2 — Método QR com Shift de Wilkinson e Deflação

A partir de H, iniciamos o processo iterativo QR com shift de Wilkinson. A cada iteração, aplicamos o shift μ, realizamos uma fatoração QR, reconstruímos a matriz $A^{(k+1)}$

via $RQ + \mu I$ e monitoramos o elemento subdiagonal $h_{m,m-1}$ para determinar a deflação, reduzindo a submatriz ativa.

A seguir listamos as matrizes obtidas em cada passo significativo do processo.

Iteração 1 (submatriz 5×5, shift $\mu = 5.644214 \cdot 10^0$):

$$\begin{bmatrix} -4.029963{\cdot}10^{-1} & 1.991235{\cdot}10^{0} & 0 & 0 & 0 \\ 1.991235{\cdot}10^{0} & 5.449236{\cdot}10^{-1} & -4.292203{\cdot}10^{-1} & 0 & 0 \\ 0 & -4.292203{\cdot}10^{-1} & 3.153779{\cdot}10^{0} & -8.398443{\cdot}10^{-1} & 0 \\ 0 & 0 & -8.398443{\cdot}10^{-1} & 5.379032{\cdot}10^{0} & -5.052555{\cdot}10^{-2} \\ 0 & 0 & 0 & -5.052555{\cdot}10^{-2} & 5.635261{\cdot}10^{0} \end{bmatrix}.$$

Iteração 2 (submatriz 5×5, shift $\mu = 5.644864 \cdot 10^0$):

$$\begin{bmatrix} -1.493263{\cdot}10^{0} & 1.326976{\cdot}10^{0} & 0 & 0 & 0 \\ 1.326976{\cdot}10^{0} & 1.568672{\cdot}10^{0} & -2.607501{\cdot}10^{-1} & 0 & 0 \\ 0 & -2.607501{\cdot}10^{-1} & 2.934708{\cdot}10^{0} & 1.781420{\cdot}10^{-2} & 0 \\ 0 & 0 & 1.781420{\cdot}10^{-2} & 5.673293{\cdot}10^{0} & -4.418222{\cdot}10^{-2} \\ 0 & 0 & 0 & -4.418222{\cdot}10^{-2} & 5.626590{\cdot}10^{0} \end{bmatrix}.$$

Iteração 3 (submatriz 5×5, shift $\mu = 5.599968 \cdot 10^0$):

$$\begin{bmatrix} -1.869430{\cdot}10^{0} & 6.854675{\cdot}10^{-1} & 0 & 0 & 0 \\ 6.854675{\cdot}10^{-1} & 1.913913{\cdot}10^{0} & -1.847281{\cdot}10^{-1} & 0 & 0 \\ 0 & -1.847281{\cdot}10^{-1} & 2.965517{\cdot}10^{0} & -5.781561{\cdot}10^{-4} & 0 \\ 0 & 0 & -5.781561{\cdot}10^{-4} & 5.700000{\cdot}10^{0} & 1.919387{\cdot}10^{-5} \\ 0 & 0 & 0 & 1.919387{\cdot}10^{-5} & 5.600000{\cdot}10^{0} \end{bmatrix}.$$

Iteração 4 (submatriz 5×5, shift $\mu = 5.600000 \cdot 10^0$):

$$\begin{bmatrix} -1.962593{\cdot}10^{0} & 3.301534{\cdot}10^{-1} & 0 & 0 & 0 \\ 3.301534{\cdot}10^{-1} & 1.990795{\cdot}10^{0} & -1.340501{\cdot}10^{-1} & 0 & 0 \\ 0 & -1.340501{\cdot}10^{-1} & 2.981798{\cdot}10^{0} & 2.205330{\cdot}10^{-5} & 0 \\ 0 & 0 & 2.205330{\cdot}10^{-5} & 5.700000{\cdot}10^{0} & 2.186108{\cdot}10^{-16} \\ 0 & 0 & 0 & -9.082539{\cdot}10^{-19} & 5.600000{\cdot}10^{0} \end{bmatrix}.$$

A partir desse ponto, o elemento subdiagonal foi inferior à tolerância, identificando o autovalor 5.6 e reduzindo a submatriz ativa para 4×4.

Iteração 5 (submatriz 4×4, shift $\mu = 5.700000 \cdot 10^0$):

$$\begin{bmatrix} -1.983665{\cdot}10^0 & 1.590131{\cdot}10^{-1} & 0 & 0 & 0 \\ 1.590131{\cdot}10^{-1} & 2.003431{\cdot}10^0 & -9.839870{\cdot}10^{-2} & 0 & 0 \\ 0 & -9.839870{\cdot}10^{-2} & 2.990235{\cdot}10^0 & -1.772961{\cdot}10^{-17} & 0 \\ 0 & 0 & 2.608740{\cdot}10^{-18} & 5.700000{\cdot}10^0 & 0 \\ 0 & 0 & 0 & 0 & 5.600000{\cdot}10^0 \end{bmatrix}.$$

Iteração 6 (submatriz 3×3, shift $\mu = 2.999951 \cdot 10^0$):

$$\begin{bmatrix} -1.989747{\cdot}10^0 & 3.175749{\cdot}10^{-2} & 0 & 0 & 0 \\ 3.175749{\cdot}10^{-2} & 1.999747{\cdot}10^0 & -4.889046{\cdot}10^{-6} & 0 & 0 \\ 0 & -4.889046{\cdot}10^{-6} & 3.000000{\cdot}10^0 & 0 & 0 \\ 0 & 0 & 0 & 5.700000{\cdot}10^0 & 0 \\ 0 & 0 & 0 & 0 & 5.600000{\cdot}10^0 \end{bmatrix}.$$

Iteração 7 (submatriz 3×3, shift $\mu = 3.000000 \cdot 10^0$):

$$\begin{bmatrix} -1.989990{\cdot}10^0 & 6.364614{\cdot}10^{-3} & 0 & 0 & 0 \\ 6.364614{\cdot}10^{-3} & 1.999990{\cdot}10^0 & 1.450727{\cdot}10^{-16} & 0 & 0 \\ 0 & -2.287119{\cdot}10^{-20} & 3.000000{\cdot}10^0 & 0 & 0 \\ 0 & 0 & 0 & 5.700000{\cdot}10^0 & 0 \\ 0 & 0 & 0 & 0 & 5.600000{\cdot}10^0 \end{bmatrix}.$$

Iteração 8 (submatriz 2×2, shift $\mu = 2.000000 \cdot 10^0$):

$$\begin{bmatrix} -1.990000{\cdot}10^0 & 3.138845{\cdot}10^{-16} & 0 & 0 & 0 \\ 1.524625{\cdot}10^{-20} & 2.000000{\cdot}10^0 & 0 & 0 & 0 \\ 0 & 0 & 3.000000{\cdot}10^0 & 0 & 0 \\ 0 & 0 & 0 & 5.700000{\cdot}10^0 & 0 \\ 0 & 0 & 0 & 0 & 5.600000{\cdot}10^0 \end{bmatrix}.$$

Matriz diagonal final:

$$A_k = \begin{bmatrix} -1.99 & 0 & 0 & 0 & 0 \\ 0 & 2.00 & 0 & 0 & 0 \\ 0 & 0 & 3.00 & 0 & 0 \\ 0 & 0 & 0 & 5.70 & 0 \\ 0 & 0 & 0 & 0 & 5.60 \end{bmatrix}.$$

Portanto, os autovalores encontrados são:

$$\{-1.99,\ 2,\ 3,\ 5.7,\ 5.6\}.$$

Os valores coincidem com os autovalores desejados, confirmando a eficiência do método de redução para Hessenberg seguido pelo método QR com shift de Wilkinson e deflação.

A tabela seguinte resume os shifts μ informado pelo código e a dimensão da submatriz ativa m em cada iteração:

Tabela 7 – Tabela de Convergência do Método QR com Shift de Wilkinson

Iteração k	μ **(shift)**	m
1	$5.644214 \cdot 10^0$	5
2	$5.644864 \cdot 10^0$	5
3	$5.599968 \cdot 10^0$	5
4	$5.600000 \cdot 10^0$	5
5	$5.700000 \cdot 10^0$	4
6	$2.999951 \cdot 10^0$	3
7	$3.000000 \cdot 10^0$	3
8	$2.000000 \cdot 10^0$	2

6 Conclusão

Neste trabalho, estudamos em detalhe os principais métodos numéricos para o cálculo de autovalores de matrizes reais, partindo de fundamentos teóricos até a implementação completa de algoritmos. Estabeleceu-se o suporte teórico necessário para compreender as técnicas mais avançadas apresentadas na sequência. A apresentação seguiu os desenvolvimentos clássicos encontrados em referências como (GOLUB; LOAN, 2013) e (TREFETHEN; BAU, 1997).

No capítulo dedicado à triangularização de matrizes, analisamos as reflexões de Householder e sua aplicação à redução de uma matriz geral à forma de Hessenberg. Essa etapa é fundamental para acelerar algoritmos iterativos, tornando-os mais eficientes sem alterar o espectro da matriz. A formulação e a implementação modular mostraram como o pré-processamento reduz significativamente o custo computacional das iterações subsequentes.

Em seguida, o estudo aprofundou-se na Fase QR, abordando métodos iterativos para obtenção de autovalores: quociente de Rayleigh, iteração da potência, potência inversa, potência inversa com *shift* e iteração simultânea. O destaque foi dado ao método QR com *shift*. Além disso, apresentamos o mecanismo de deflação, que reduz progressivamente o tamanho da submatriz ativa, permitindo extrair autovalores de maneira estável e eficiente.

A parte final do trabalho reuniu as etapas anteriores em uma implementação completa das Fases 1 e 2, aplicadas a um exemplo numérico não trivial. A matriz escolhida, construída a partir de autovalores prescritos e perturbada para aproximar situações reais, mostrou como o método utilizado se comporta em casos desafiadores. As iterações detalhadas evidenciaram a estabilidade do processo, o papel dos *shifts* e a rapidez com que a deflação identifica autovalores convergidos. O resultado final exibiu a diagonalização numérica da matriz em conformidade com o comportamento teórico esperado.

Além disso, este trabalho dá continuidade direta ao estudo desenvolvido no TCC 1 (dos Santos, 2025). No trabalho anterior foram apresentados os fundamentos essenciais de Álgebra Linear Numérica, incluindo normas, estabilidade numérica, decomposição em valores singulares (SVD) e métodos de fatoração QR, além de comparações computacionais entre diferentes variantes desses algoritmos. Esses resultados formaram a base teórica e prática necessária para o desenvolvimento do presente estudo.

Em síntese, este TCC apresentou tanto os fundamentos matemáticos quanto a realização computacional dos métodos iterativos. Concluímos, portanto, que o método QR com shift de Wilkinson, aliado à redução inicial à forma de Hessenberg, constitui uma estratégia eficiente e estável para o cálculo de autovalores reais.

Referências

dos Santos, A. B. O. *Decomposição em Valores Singulares, Métodos e Aplicações*. Dissertação (Monografia (Bacharelado em Matemática)) — Universidade Federal de São Carlos, São Carlos – SP, 2025. Orientador: Prof. Dr. Sávio Brochini Rodrigues. Citado 2 vezes nas páginas 18 e 49.

GOLUB, G. H.; LOAN, C. F. V. *Matrix Computations*. 4. ed. [S.l.]: Johns Hopkins University Press, 2013. Citado 2 vezes nas páginas 9 e 49.

HIGHAM, N. J. *Accuracy and Stability of Numerical Algorithms*. 2. ed. [S.l.]: SIAM, 2002. Citado na página 9.

PANJU, M. *Iterative Methods for Computing Eigenvalues and Eigenvectors*. 2011. ArXiv preprint arXiv:1105.1185. ⟨https://arxiv.org/abs/1105.1185⟩. Citado na página 23.

STRANG, G. *Introduction to Linear Algebra*. 6. ed. Wellesley, MA: Wellesley-Cambridge Press, 2023. ISBN 978-1733146697. Citado na página 9.

TREFETHEN, L. N.; BAU, D. *Numerical Linear Algebra*. [S.l.]: SIAM, 1997. Citado 4 vezes nas páginas 9, 18, 23 e 49.

A Código método da potência

```python
import numpy as np

# --- Funcao para construir matriz simetrica com autovalores dados ---
def build_symmetric_with_eigs(eigs, seed=None):
"""
eigs: lista/array de autovalores (lambda_1,...,lambda_n)
retorna A = Q diag(eigs) Q^T com Q ortogonal gerada por QR de matriz aleatoria
"""
if seed is not None:
np.random.seed(seed)
n = len(eigs)
# gera matriz aleatoria e obtem Q pela fatoracao QR
X = np.random.randn(n, n)
Q, _ = np.linalg.qr(X)
D = np.diag(eigs)
A = Q @ D @ Q.T
# forca simetria numerica
A = (A + A.T) / 2
return A

# --- Exemplo: autovalores com razao proxima de 1 (convergencia lenta) ---
#eigs = [1.0, 0.995, 0.5, 0.3, 0.1]   # lambda1=1.0, lambda2=0.995 -> razao 0.995
eigs = [5.7, 5.6, 3, 2]
A = build_symmetric_with_eigs(eigs, seed=42)

# Checar autovalores (ordenados por modulo decrescente) e a razao
vals, vecs = np.linalg.eig(A)
# ordenar por valor absoluto decrescente
idx = np.argsort(-np.abs(vals))
vals_sorted = vals[idx]
ratio = abs(vals_sorted[1]) / abs(vals_sorted[0])

print("Autovalores (ordenados por |.| decrescente):")
```

```
for i, ev in enumerate(vals_sorted, start=1):
print(f"    {i} = {ev:.6f}")
print(f"\nRazao | 2 |/| 1 | = {ratio:.6f}")

if __name__ == "__main__":
lam, v, it, history = power_iteration(A=A, tol=1e-12,
max_iter=100, return_history=True)
residuo = np.linalg.norm(A @ v - lam * v)

print(f"Autovalor dominante     {lam:.10f}")
print("Autovetor (||v||=1):", v)
print(f"Numero de itera  es: {it}")
print(f"Erro residual (||A v -    v||   ) = {residuo:.2e}")

# ---------- Construcao da tabela ----------
history = np.array(history)
vet_erros = np.abs(history - lam)

# Razao de erros (entre erros sucessivos)
razao_erros = np.zeros_like(vet_erros)
razao_erros[1:] = vet_erros[1:] / vet_erros[:-1]

history = history[:-2]
vet_erros = vet_erros[:-2]
razao_erros = razao_erros[:-2]

# Criacao do DataFrame
tabela = pd.DataFrame({
    "Iteracao": np.arange(1, len(history) + 1),
    "Erro | _k  -  _final |": vet_erros,
    "Razao de erros": razao_erros
})

pd.options.display.float_format = '{:.2e}'.format
print("\nTabela de convergencia:\n")
print(tabela.round(6).to_string(index=False))
```

B Código método da potência inversa

```
import numpy as np
from scipy.linalg import lu_factor, lu_solve

def inverse_iteration(A, mu, v0=None, tol=1e-12,
max_iter=1000, return_history=False):

"""
Iteracao inversa com shift fixo (Alg. 27.2) para matriz
(preferencialmente simetrica) A.
Parametros
----------
A : (n,n) array_like
mu : float
Shift alvo; o metodo convergira para o autovetor cujo
autovalor e mais proximo de mu.
v0 : (n,) array_like or None
Chute inicial (normalizado internamente). Se None, usa
aleatorio.
tol : float
Crit rio de parada: ||A v - lambda v||_2 <= tol * ||A||_2.
max_iter : int
return_history : bool
Se True, retorna tambem a lista de lambdas aproximados.

Retorna
-------
lam : float          # autovalor aproximado (quociente de
Rayleigh)
v   : (n,) ndarray   # autovetor unit rio
k   : int            # itera  es realizadas
hist: list[float]    # (opcional) historico de lambdas
"""
A = np.array(A, dtype=float)
n = A.shape[0]

# vetor inicial
if v0 is None:
v = np.random.rand(n)
```

```
34      else:
35      v = np.array(v0, dtype=float)
36      v /= np.linalg.norm(v)
37
38      I = np.eye(n)
39      hist = []
40
41      # Fatoracao LU de (A - mu*I)
42      try:
43      lu, piv = lu_factor(A - mu * I)
44      except np.linalg.LinAlgError:
45      eps = 1e-14 * np.linalg.norm(A, 2)
46      lu, piv = lu_factor(A - (mu + eps) * I)
47
48      for k in range(1, max_iter + 1):
49      w = lu_solve((lu, piv), v)
50      v = w / np.linalg.norm(w)
51      lam = float(v @ (A @ v))
52      hist.append(lam)
53
54      res = np.linalg.norm(A @ v - lam * v)
55      if res <= tol:
56      return (lam, v, k, hist) if return_history else (lam, v, k)
57
58      return (lam, v, max_iter, hist) if return_history else (lam,
        v, max_iter)
59
60
61      # ----------------- Exemplo de uso -----------------
62      # --- Funcao para construir matriz simetrica com autovalores
        dados ---
63      def build_symmetric_with_eigs(eigs, seed=None):
64      """
65      eigs: lista/array de autovalores (lambda_1,...,lambda_n)
66      retorna A = Q diag(eigs) Q^T com Q ortogonal gerada por QR
        de matriz aleatoria
67      """
68      if seed is not None:
69      np.random.seed(seed)
70      n = len(eigs)
71      # gera matriz aleatoria e obtem Q pela fatoracao QR
72      X = np.random.randn(n, n)
```

```
73      Q, _ = np.linalg.qr(X)
74      D = np.diag(eigs)
75      A = Q @ D @ Q.T
76      # forca simetria numerica
77      A = (A + A.T) / 2
78      return A
79
80  # --- Exemplo: autovalores com razqo proxima de 1
    (convergencia lenta) ---
81  #eigs = [1.0, 0.995, 0.5, 0.3, 0.1]   # lambda1=1.0,
    lambda2=0.995 -> razao 0.995
82  eigs = [5.7, 5.6, 3, 2]
83  A = build_symmetric_with_eigs(eigs, seed=42)
84
85  # Checar autovalores (ordenados por m dulo decrescente) e a
    raz o
86  vals, vecs = np.linalg.eig(A)
87  # ordenar por valor absoluto decrescente
88  idx = np.argsort(-np.abs(vals))
89  vals_sorted = vals[idx]
90  ratio = abs(vals_sorted[1]) / abs(vals_sorted[0])
91
92  print("Autovalores (ordenados por |.| decrescente):")
93  for i, ev in enumerate(vals_sorted, start=1):
94      print(f"    {i} = {ev:.6f}")
95  print(f"\nRaz o | 2 |/| 1 | = {ratio:.6f}")
96
97  if __name__ == "__main__":
98      mu = 5.58
99
100     lam, v, it, hist = inverse_iteration(A, mu, tol=1e-12,
    max_iter=1000, return_history=True)
101     residuo = np.linalg.norm(A @ v - lam * v)
102
103     print(f"Shift mu = {mu}")
104     print(f"Autovalor aproximado: {lam:.10f}")
105     print(f"Autovetor (norma 1): {v}")
106     print(f"Itera  es: {it}")
107     print(f"Erro residual ||Av -  v ||_2 = {residuo:.2e}")
108
109     # ----------------- Criacao da tabela -----------------
110     import pandas as pd
```

```
111
112     # Remove a ultima iteracao (nao queremos incluir a final)
113     hist_sem_ultima = hist[:-1]
114     erros = [abs(l - lam) for l in hist_sem_ultima]
115
116     # Diferen a entre lambdas
117     diffs = [l - lam for l in hist_sem_ultima]
118
119     # Razoes entre erros consecutivos
120     razoes = [erros[i+1]/erros[i] if erros[i] != 0 else np.nan
        for i in range(len(erros)-1)]
121     razoes.append(np.nan)  # ultimo nao tem razao
122
123     tabela = pd.DataFrame({
124         "Itera  o": np.arange(1, len(hist_sem_ultima)+1),
125         " _k  -   ": diffs,
126         "Raz o de erros": razoes
127     })
128
129     print("\nTabela de Convergencia:")
130     print(tabela.to_string(index=False))
```

C Código iteração do quociente de Rayleigh

```
import numpy as np
import pandas as pd

def rayleigh_quotient_iteration(A, v0=None, tol=1e-12,
max_iter=1000, return_history=False):
"""
Implementa o Algoritmo 27.3 (Rayleigh Quotient Iteration)
para matriz simetrica A.
Retorna (autovalor aproximado, autovetor normalizado, numero
de iteracoes [, historico]).
"""
n = A.shape[0]
A = np.array(A, dtype=float)

if v0 is None:
v = np.random.rand(n)  #vai para o autoval 2
# v = np.array([-1.0,  1,  1,  1])  #vai para o 5.6
# v = np.array([-1.0,  -1,  1,  1])  #vai para o 3.0
#  v = np.array([-1.0,  -1,  -1,  1])
else:
v = np.array(v0, dtype=float)

v /= np.linalg.norm(v)
lam = v.T @ A @ v

hist = [lam]  # hist rico dos   (shifts   )
for k in range(max_iter):
try:
w = np.linalg.solve(A - lam * np.eye(n), v)
except np.linalg.LinAlgError:
print(f"Matriz singular na itera  o {k}")
break

v = w / np.linalg.norm(w)
lam_new = float(v.T @ A @ v)

hist.append(lam_new)

```

```
36      if abs(lam_new - lam) < tol:
37      return (lam_new, v, k + 1, hist) if return_history else
        (lam_new, v, k + 1)
38
39      lam = lam_new
40
41      return (lam, v, max_iter, hist) if return_history else (lam,
        v, max_iter)
42
43
44      # ----------------- Funcao auxiliar -----------------
45      def build_symmetric_with_eigs(eigs, seed=None):
46      """
47      eigs: lista/array de autovalores (lambda_1,...,lambda_n)
48      retorna A = Q diag(eigs) Q^T com Q ortogonal gerada por QR
        de matriz aleat ria
49      """
50      if seed is not None:
51      np.random.seed(seed)
52      n = len(eigs)
53      X = np.random.randn(n, n)
54      Q, _ = np.linalg.qr(X)
55      D = np.diag(eigs)
56      A = Q @ D @ Q.T
57      A = (A + A.T) / 2  # forca simetria
58      return A
59
60
61      # ----------------- Exemplo de uso -----------------
62      eigs = [5.7, 5.6, 3, 2]  # autovalores desejados
63      A = build_symmetric_with_eigs(eigs, seed=42)
64
65      # Verificacao dos autovalores
66      vals, _ = np.linalg.eig(A)
67      idx = np.argsort(-np.abs(vals))
68      vals_sorted = vals[idx]
69      ratio = abs(vals_sorted[1]) / abs(vals_sorted[0])
70
71      print("Autovalores (ordenados por |.| decrescente):")
72      for i, ev in enumerate(vals_sorted, start=1):
73      print(f"    {i} = {ev:.6f}")
74      print(f"\nRaz o | 2 |/| 1 | = {ratio:.6f}\n")
```

```
75
76
77     # ---------- Execucao da iteracao ----------
78     lam, v, it, hist = rayleigh_quotient_iteration(A, tol=1e-12,
       max_iter=10, return_history=True)
79     residuo = np.linalg.norm(A @ v - lam * v)
80
81     print(f"Autovalor aproximado: {lam:.10f}")
82     print(f"Autovetor aproximado: {v}")
83     print(f"Itera  es: {it}")
84     print(f"Erro residual ||Av -  v || = {residuo:.2e}")
85
86     # ---------- Construcao da tabela ----------
87     hist_sem_ultima = hist[:-1]
88     erros = [abs(l - lam) for l in hist_sem_ultima]
89     diffs = [l - lam for l in hist_sem_ultima]
90
91     # Razao de erros |E_{k+1}| / |E_k|
92     razoes = [erros[i+1] / erros[i] if erros[i] != 0 else np.nan
       for i in range(len(erros)-1)]
93     razoes.append(np.nan)
94
95     # Calculo de p     log(E_{k+1}/E_k) / log(E_k/E_{k-1})
96     p_vals = [np.nan, np.nan]
97     for i in range(2, len(erros)-1):
98     if erros[i-1] != 0 and erros[i] != 0 and erros[i+1] != 0:
99     num = np.log(erros[i+1] / erros[i])
100    den = np.log(erros[i] / erros[i-1])
101    p_vals.append(num / den)
102    else:
103    p_vals.append(np.nan)
104    p_vals.append(np.nan)
105
106    tabela = pd.DataFrame({
107        "Itera  o": np.arange(1, len(hist_sem_ultima)+1),
108        "   (shift)": hist_sem_ultima,
109        " _k  -   ": diffs,
110        "Razao de erros": razoes,
111        "p": p_vals
112    })
113
114    print("\nTabela de Convergencia:")
```

```
115     print(tabela.to_string(index=False))
```

D Código QR sem shifts

```
import numpy as np
import pandas as pd

def pure_qr_algorithm_with_convergence(A, max_iter=15, lambda_ref_index=0):
"""
Implementa o QR 'puro' (sem shifts) e registra a convergencia
do autovalor principal (A[0,0]) em cada iteracao.
"""
A = np.array(A, dtype=float)
n = A.shape[0]

# Q_total nao e estritamente necessario para apenas autovalores no QR puro
# Q_total = np.eye(n)

convergence_data = []
lambda_k_prev = np.diag(A)[lambda_ref_index] # Inicializa com o primeiro autovalor aproximado

for k in range(1, max_iter + 1):
Q, R = np.linalg.qr(A)
A = R @ Q
# Q_total = Q_total @ Q # Autovetores

# Autovalores na diagonal de A_k (autovalores aproximados)
eigvals_k = np.diag(A)

# O autovalor 'dominante' esta na posicao A[0,0] se ele for o maior
lambda_k = eigvals_k[lambda_ref_index]

# Medida de Convergencia (diferen a entre itera  es)
# O erro na imagem e |lambda_k - lambda_exato|
# Como nao temos o lambda_exato, usamos a diferenca entre iteracoes: |lambda_k - lambda_{k-1}|
erro_lambda = abs(lambda_k - lambda_k_prev)

```

```
34     # Razao de Erros: erro_k / erro_{k-1}
35     razao_erros = np.nan
36     if k > 1 and erro_lambda_prev != 0:
37     razao_erros = erro_lambda / erro_lambda_prev
38 
39     # Registra (a iteracao e deslocada de 1, pois lambda_k_prev
       e da iteracao k-1)
40     if k > 1:
41     # Usamos lambda_k_prev (erro da iteracao anterior) para
       coincidir com a logica da imagem
42     convergence_data.append({
43         'Iteracao': k - 1,
44         'lambda_k - lambda': erro_lambda_prev,
45         'Razao de erros': razao_erros
46     })
47 
48     erro_lambda_prev = erro_lambda
49     lambda_k_prev = lambda_k
50 
51     # Criterio de parada (se desejar)
52     # if np.allclose(np.tril(A, -1), 0, atol=1e-12) and k > 1:
53     #     break
54 
55     # Autovalores finais ordenados por magnitude decrescente
56     final_eigvals = np.diag(A)
57     final_eigvals = np.sort(np.abs(final_eigvals))[::-1]
58 
59     # O autovalor aproximado da imagem e o segundo maior (5.6,
       proximo ao 5.587)
60     # Autovalor aproximado final: A[1,1]
61     lambda_approx_final = np.diag(A)[1]
62 
63     # Autovetor: O QR puro calcula todos os autovetores
       (Q_total, se calculado)
64     # Aqui, vamos ignorar o calculo do autovetor para manter o
       foco no QR *puro*.
65     # Se fosse necessario, seria np.linalg.eig(A)[1] para o
       autovetor exato.
66 
67     # Tabela de Convergencia
68     convergence_df = pd.DataFrame(convergence_data)
69 
```

```
70     # Formata  o da tabela (com 15 itera  es , teremos 14
       linhas de erro)
71     if not convergence_df.empty:
72     convergence_df['lambda_k - lambda'] =
       convergence_df['lambda_k - lambda'].apply(lambda x:
       f'{x:.7e}')
73     convergence_df['Razao de erros'] = convergence_df['Razao de
       erros'].round(6)
74
75     return final_eigvals, lambda_approx_final, max_iter,
       convergence_df
76
77     # ----------------- Funcao auxiliar -----------------
78     def build_symmetric_with_eigs(eigs, seed=None):
79     """
80     eigs: lista/array de autovalores (lambda_1,...,lambda_n)
81     retorna A = Q diag(eigs) Q^T com Q ortogonal gerada por QR
       de matriz aleatoria
82     """
83     if seed is not None:
84     np.random.seed(seed)
85     n = len(eigs)
86     X = np.random.randn(n, n)
87     Q, _ = np.linalg.qr(X)
88     D = np.diag(eigs)
89     A = Q @ D @ Q.T
90     A = (A + A.T) / 2  # forca simetria
91     return A
92
93
94     # ----------------- Exemplo de uso -----------------
95     eigs = [5.7, 5.6, 3, 2]  # autovalores desejados
96     A = build_symmetric_with_eigs(eigs, seed=42)
97
98
99     eigvals, lambda_approx, iters, conv_table =
       pure_qr_algorithm_with_convergence(A, max_iter=100,
       lambda_ref_index=1)
100
101    # Calculo dos autovalores exatos e razao para a saida
102    exact_eigvals = np.linalg.eigvals(A)
103    exact_eigvals_sorted = np.sort(np.abs(exact_eigvals))[::-1]
```

```
104
105
106     print("Autovalores (ordenados por |.| decrescente):")
107     for i, val in enumerate(exact_eigvals_sorted):
108     # A matriz A s  tem 3 autovalores, mas a imagem tem 4.
109     print(f"  {i+1} = {val:.6f}")
110     #if i == 2: break # Limita aos 3 autovalores da matriz A
111
112     # A razao | 2 |/| 1 | do QR puro e importante para a
        velocidade de convergencia
113     ratio_lambda = abs(exact_eigvals_sorted[1] /
        exact_eigvals_sorted[0])
114     print(f"\nRazao | 2 |/| 1 | = {ratio_lambda:.6f}")
115     print(f"Autovalor aproximado: {lambda_approx:.10f}")
116     # Autovetor n o    calculado diretamente (seria necess rio
        Q_total)
117     print(f"Itera cos: {iters}")
118     # O erro residual para o QR puro seria para A_final - D
        (D=matriz diagonal)
119     print(f"Erro residual ||Av -  v ||_2 = [N o calculado no QR
        Puro]")
120
121
122     print("\nTabela de Converg ncia:")
123     print("\nTabela de Converg ncia:")
124     print(conv_table.to_string(index=False))
```

E Código QR com shifts

```
1   .
```

F Exemplo “fácil”

```
import numpy as np
import pandas as pd

def householder_tridiagonal(A):
A = np.array(A, dtype=float, copy=True)
n = A.shape[0]
Q = np.eye(n)

for k in range(n-2):
x = A[k+1:, k]
normx = np.linalg.norm(x)
if normx == 0.0:
continue
sign = 1.0 if x[0] >= 0 else -1.0
u1 = x[0] + sign * normx
v = x.copy()
v[0] = u1
v /= np.linalg.norm(v)

A[k+1:, k:] -= 2.0 * np.outer(v, v @ A[k+1:, k:])
A[:, k+1:] -= 2.0 * np.outer(A[:, k+1:] @ v, v)
Q[:, k+1:] -= 2.0 * np.outer(Q[:, k+1:] @ v, v)

A[k+2:, k] = 0.0
A[k, k+2:] = 0.0

T = np.triu(A) + np.triu(A, 1).T
return T, Q


def wilkinson_shift_2x2(a, b, c):
if b == 0.0:
return c
delta = 0.5 * (a - c)
return c - np.sign(delta if delta != 0.0 else 1.0) * (b*b) /
(abs(delta) + np.hypot(delta, b))


```

```
38 def practical_qr_symmetric(A, tol=1e-12, max_iters=10_000,
   use_wilkinson=True):
39 n = A.shape[0]
40 T, Q = householder_tridiagonal(A)
41 m = n
42 it = 0
43
44 # Coletas extras
45 shifts = []
46 errors = []
47 error_ratio = []
48 max_offdiag = []
49 frob_norm = []
50 # MODIFICACAO 1: Adicionar lista para armazenar 'm'
51 m_history = []
52
53 prev_error = None
54
55 while m > 1 and it < max_iters:
56 # MODIFICACAO 2: Capturar o 'm' atual no inicio da iteracao
57 current_m = m
58
59 # Coletar metricas antes da iteracao
60 off = T - np.diag(np.diag(T))
61 #max_offdiag.append(float(np.max(np.abs(off))))
62 #frob_norm.append(float(np.linalg.norm(off, ord='fro')))
63
64 if abs(T[m-1, m-2]) <= tol*(abs(T[m-1, m-1]) + abs(T[m-2,
   m-2])):
65 T[m-1, m-2] = T[m-2, m-1] = 0.0
66 m -= 1
67 continue
68 else:
69 max_offdiag.append(float(np.max(np.abs(off))))
70 frob_norm.append(float(np.linalg.norm(off, ord='fro')))
71
72 if use_wilkinson and m >= 2:
73 a = T[m-2, m-2]
74 b = T[m-2, m-1]
75 c = T[m-1, m-1]
76 mu = wilkinson_shift_2x2(a, b, c)
77 else:
```

```
mu = T[m-1, m-1]

Qk, Rk = np.linalg.qr(T[:m, :m] - mu*np.eye(m))
T[:m, :m] = Rk @ Qk + mu*np.eye(m)
Q[:, :m] = Q[:, :m] @ Qk

shifts.append(float(mu))
errors.append(float(abs(T[m-1, m-2])))
# MODIFICACAO 3: Adicionar o 'm' capturado
m_history.append(current_m)

if prev_error is None:
error_ratio.append(np.nan)
else:
error_ratio.append(float(abs(T[m-1, m-2]) / prev_error))
prev_error = abs(T[m-1, m-2])

it += 1

if it == max_iters:
raise RuntimeError("Numero maximo de iteracoes atingido.")

w = np.diag(T).copy()
idx = np.argsort(w)
# MODIFICACAO 4: Retornar m_history
return w[idx], Q[:, idx], shifts, errors, error_ratio, max_offdiag, frob_norm, m_history


if __name__ == "__main__":
np.set_printoptions(precision=6, suppress=True)
n = 25
seed = 42

# A fun  o orthonormal_random nao esta definida, mas o exemplo FACIL
# gera a matriz D_easy e a constroi com uma Qrand gerada localmente.

# --------------- Exemplo FACIL ----------------
# lambda_k = (-1)^k / 2^k, k=1..n
```

```
    lambdas_easy = np.array([((-1)**(k+1)) / (2**(k+1)) for k in range(n)])
    D_easy = np.diag(lambdas_easy)

    # Generating a random orthogonal matrix
    rng = np.random.default_rng(seed)
    X = rng.normal(size=(n, n))
    Qrand, _ = np.linalg.qr(X)

    A_easy = Qrand @ D_easy @ Qrand.T

    # aplica fase 1 (Householder)   obtem tridiagonal T_easy
    T_easy, Qh_easy = householder_tridiagonal(A_easy)

    # roda o QR pr tico (fase 2)
    # MODIFICACAO 5: Capturar m_history na chamada da fun  o
    vals_easy, vecs_easy, shifts, errors, ratio, maxOD, frob, m_history = practical_qr_symmetric(A_easy, tol=1e-12, use_wilkinson=True)

    # verificacao/residuos
    residual_easy = np.linalg.norm(A_easy @ vecs_easy - vecs_easy * vals_easy, ord=np.inf)

    print("=== EXEMPLO F CIL ===")
    print("Primeiros 8 autovalores (D):", lambdas_easy[:8])
    print("Autovalores (obtidos)   5 primeiros:", vals_easy[:5])
    print("Erro residual ||A V - V  ||_inf =", residual_easy)
    print("Raz es |lambda_{i+1}/lambda_i| (primeiros 8):",
    np.abs(lambdas_easy[1:8] / lambdas_easy[:7]))


    # Tabela pandas da converg ncia
    # MODIFICACAO 6: Usar 'm_history' no lugar de 'ratio' e mudar o nome da coluna
    df = pd.DataFrame({
        "Itera  o": range(1, len(shifts)+1),
        "  (shift)": shifts,
        "| _k  -   |": errors,
        "Valor de m": m_history
    })
```

```
152     pd.set_option("display.float_format", "{:.3e}".format)
153     # Mudar o formato de m_history para inteiro
154     df['Valor de m'] = df['Valor de m'].astype(int)
155 
156 
157     print("\nTabela de Convergencia:")
158     print(df)
159 
160     # Tabela final das normas
161     df2 = pd.DataFrame({
162         "Max fora diagonal": maxOD,
163         "Norma Frobenius off": frob
164     })
165     print("\nTabela da norma de Frobenius e maximo fora da
        diagonal:")
166     print(df2)
167 
168     import matplotlib.pyplot as plt
169     frob_log10 = np.log10(frob_easy_case)
170 
171     # Assuming 'frob_easy_case' from the previous cell's
        execution contains the Frobenius norm data
172     plt.figure(figsize=(10, 6))
173     plt.plot(frob_log10, marker='o', linestyle='-', color='blue')
174     #plt.plot(frob, marker='o', linestyle='-')
175     plt.xlabel("Iteracao")
176     plt.ylabel("Log_10 da norma de Frobenius")
177     plt.title("Convergencia da Norma de Frobenius (Exemplo
        Facil)")
178     plt.grid(True)
179     plt.show()
```

G Exemplo “difícil”

```
import numpy as np
import pandas as pd

def orthonormal_random(n, seed=None):
"""
Gera uma matriz ortogonal Q via QR de uma matriz aleatoria
normal.
Ajusta sinais para obter distribuicao ortogonal 'uniforme'.
"""
rng = np.random.default_rng(seed)

X = rng.normal(size=(n, n))
Q, R = np.linalg.qr(X)
# ajusta sinais para garantir Q ortogonal "canonical"
D = np.diag(np.sign(np.diag(R)))
D[D == 0] = 1.0
Q = Q @ D
return Q


def householder_tridiagonal(A):
A = np.array(A, dtype=float, copy=True)
n = A.shape[0]
Q = np.eye(n)

for k in range(n-2):
x = A[k+1:, k]
normx = np.linalg.norm(x)
if normx == 0.0:
continue
sign = 1.0 if x[0] >= 0 else -1.0
u1 = x[0] + sign * normx
v = x.copy()
v[0] = u1
v /= np.linalg.norm(v)

A[k+1:, k:] -= 2.0 * np.outer(v, v @ A[k+1:, k:])
A[:, k+1:] -= 2.0 * np.outer(A[:, k+1:] @ v, v)
```

```
38      Q[:, k+1:] -= 2.0 * np.outer(Q[:, k+1:] @ v, v)
39
40      A[k+2:, k] = 0.0
41      A[k, k+2:] = 0.0
42
43      T = np.triu(A) + np.triu(A, 1).T
44      return T, Q
45
46
47      def wilkinson_shift_2x2(a, b, c):
48      if b == 0.0:
49      return c
50      delta = 0.5 * (a - c)
51      return c - np.sign(delta if delta != 0.0 else 1.0) * (b*b) /
        (abs(delta) + np.hypot(delta, b))
52
53
54      def practical_qr_symmetric(A, tol=1e-12, max_iters=10_000,
        use_wilkinson=True):
55      n = A.shape[0]
56      T, Q = householder_tridiagonal(A)
57      m = n
58      it = 0
59
60      # Coletas extras
61      shifts = []
62      errors = []
63      error_ratio = []
64      max_offdiag = []
65      frob_norm = []
66      # MODIFICACAO 1: Adicionar lista para armazenar 'm'
67      m_history = []
68
69      prev_error = None
70
71      while m > 1 and it < max_iters:
72      # Capturar o 'm' atual no inicio da iteracao antes de ele
        ser potencialmente decrementado
73      current_m = m
74
75      # Coletar metricas antes da iteracao
76      off = T - np.diag(np.diag(T))
```

```
77      #max_offdiag.append(float(np.max(np.abs(off))))
78      #frob_norm.append(float(np.linalg.norm(off, ord='fro')))
79
80      # Verificar convergencia
81      if abs(T[m-1, m-2]) <= tol*(abs(T[m-1, m-1]) + abs(T[m-2,
        m-2])):
82      #debug: imprime o suposto autovalor
83      ##print("Autovalor encontrado:", T[m-1, m-1])
84
85      T[m-1, m-2] = T[m-2, m-1] = 0.0
86      m -= 1 # m e decrementado aqui
87      continue
88      else:
89      max_offdiag.append(float(np.max(np.abs(off))))
90      frob_norm.append(float(np.linalg.norm(off, ord='fro')))
91
92      if use_wilkinson and m >= 2:
93      a = T[m-2, m-2]
94      b = T[m-2, m-1]
95      c = T[m-1, m-1]
96      mu = wilkinson_shift_2x2(a, b, c)
97      else:
98      mu = T[m-1, m-1]
99
100     Qk, Rk = np.linalg.qr(T[:m, :m] - mu*np.eye(m))
101     T[:m, :m] = Rk @ Qk + mu*np.eye(m)
102     Q[:, :m] = Q[:, :m] @ Qk
103
104     shifts.append(float(mu))
105     errors.append(float(abs(T[m-1, m-2])))
106     m_history.append(current_m) # MODIFICACAO 2: Adicionar o 'm'
        capturado
107
108     if prev_error is None:
109     error_ratio.append(np.nan)
110     else:
111     error_ratio.append(float(abs(T[m-1, m-2]) / prev_error))
112     prev_error = abs(T[m-1, m-2])
113
114     it += 1
115
116
```

```
117     if it == max_iters:
118     raise RuntimeError("Numero maximo de iteracoes atingido.")
119 
120     w = np.diag(T).copy()
121     idx = np.argsort(w)
122     # MODIFICACAO 3: Retornar m_history
123     return w[idx], Q[:, idx], shifts, errors, error_ratio,
        max_offdiag, frob_norm, m_history
124 
125 
126 
127     # --------------- Exemplo DIFICIL ----------------
128     if __name__ == "__main__":
129     np.set_printoptions(precision=6, suppress=True)
130     n = 25
131     seed = 42
132 
133     rng = np.random.default_rng(seed)
134     X = rng.normal(size=(n, n))
135     Qrand, _ = np.linalg.qr(X)
136 
137     #Qrand = orthonormal_random(n, seed=seed)
138     #print("FROB. NORM", np.linalg.norm(Qrand @ Qrand.T -
        np.eye(n), ord='fro'))
139 
140     # lambda_k = cos(k*pi/m), k=1..m  (m=n)
141     m = n
142     lambdas_hard = np.array([np.cos((k+1) * np.pi / m) for k in
        range(n)])  # k+1 para comecar em k=1
143     ##transformei no "facil", mas nele nao calcular o os
        autovalores corretamente... o que esta diferente?
144     ##lambdas_hard = np.array([((-1)**(k+1)) / (2**(k+1)) for k
        in range(n)])
145 
146     D_hard = np.diag(lambdas_hard)
147     A_hard = Qrand @ D_hard @ Qrand.T
148 
149     # fase 1
150     T_hard, Qh_hard = householder_tridiagonal(A_hard)
151 
152     # roda QR pratico
153     # MODIFICACAO 4: Capturar m_history na chamada da funcao
```

```
154     vals_hard, vecs_hard, shifts_hard, errors_hard, ratio_hard,
        maxOD_hard, frob_hard_case, m_history_hard =
        practical_qr_symmetric(A_hard, tol=1e-12, use_wilkinson=True)
155     residual_hard = np.linalg.norm(A_hard @ vecs_hard -
        vecs_hard * vals_hard, ord=np.inf)
156
157     print("=== EXEMPLO DIFICIL ===")
158     print("Primeiros 8 autovalores (D):", lambdas_hard[:25])
159     print("Autovalores (obtidos)    5 primeiros:",
        vals_hard[:5])
160     print("Erro residual ||A V - V L||_inf =", residual_hard)
161     print("Razoes |lambda_{i+1}/lambda_i| (primeiros 8):",
162     np.abs(lambdas_hard[1:8] / lambdas_hard[:7]))
163
164     # Tabela pandas da convergencia
165     # MODIFICACAO 5: Usar 'm_history_hard' no lugar de
        'ratio_hard' e mudar o nome da coluna
166     df = pd.DataFrame({
167         "Iteracao": range(1, len(shifts_hard)+1),
168         "mu (shift)": shifts_hard,
169         "|lambda_k - lambda|": errors_hard,
170         "Valor de m": m_history_hard
171     })
172     pd.set_option("display.float_format", "{:.3e}".format)
173     # Mudar o formato de m_history_hard para inteiro
174     df['Valor de m'] = df['Valor de m'].astype(int)
175
176     print("\nTabela de Convergencia:")
177     print(df)
178
179     # Tabela final das normas
180     df2 = pd.DataFrame({
181         "Max fora diagonal": maxOD_hard,
182         "Norma Frobenius off": frob_hard_case
183     })
184
185     # Define a opcao para exibir todas as linhas
186     pd.set_option('display.max_rows', None)
187
188     try:
189     print("\nTabela da norma de Frobenius e maximo fora da
        diagonal:")
```

```
190     print(df2)
191     except NameError:
192     print("A variavel 'df2' nao foi encontrada. Por favor,
        execute a celula que a gera primeiro.")
193
194     # Define o grafico
195     import matplotlib.pyplot as plt
196     frob_log10d = np.log10(frob_hard_case)
197
198     plt.figure(figsize=(10, 6))
199     plt.plot(frob_log10d, marker='o', linestyle='-', color='red')
200     plt.xlabel("Iteracao")
201     plt.ylabel("Log_10 da norma de Frobenius")
202     plt.title("Convergencia da Norma de Frobenius (Exemplo
        Dificil)")
203     plt.grid(True)
204     plt.show()
205
206
207     # Grafico unidos exemplo facil e dificil
208     import matplotlib.pyplot as plt
209
210     # Create a new figure with a specified size
211     plt.figure(figsize=(12, 7))
212
213     # Plot 'frob_log10' (Easy Case) com cor azul
214     plt.plot(frob_log10, marker='o', linestyle='-', label='Norma
        de Frobenius Caso Facil', color='blue')
215
216     # Plot 'frob_log10d' (Difficult Case) on the same axes com
        cor laranja
217     plt.plot(frob_log10d, marker='x', linestyle='--',
        label='Norma de Frobenius Caso Dificil', color='red')
218
219     # Add title and labels
220     plt.title('Comparativo da Convergencia da Norma de
        Frobenius')
221     plt.xlabel('Iteracao')
222     plt.ylabel('Log_10 da norma de Frobenius')
223
224     # Add legend
225     plt.legend()
```

```
226
227     # Add grid for better readability
228     plt.grid(True)
229
230     # Display the plot
231     plt.show()
```

H Exemplo fases 1 e 2

```
import numpy as np

# Definir semente para resultados reprodutiveis
np.random.seed(42)

# --- Configuracao de Impressao ---
np.set_printoptions(
precision=6,
suppress=False,
formatter={'float': '{: 0.6e}'.format}
)


# --- Construcao da Matriz A (Simetrica com Autovalores
Especificos) ---
print("==============================================================")
print("Construcao da Matriz A (5x5, Simetrica)")
print("Autovalores Desejados: {5.7, 5.6, 3, 2, -1.99}")
print("==============================================================")

autovalores_desejados = np.array([5.7, 5.6, 3, 2, -1.99])
D = np.diag(autovalores_desejados)
R = np.random.rand(5, 5)
Q, _ = np.linalg.qr(R)
A = Q @ D @ Q.T

print("Matriz A (A = Q * D * Q.T):")
print(A)
print("\nVerificacao (A e simetrica?):", np.allclose(A, A.T))
print("--------------------------------------------------------------\


# --- Fase 1: Triangularizacao (Usando o Metodo QR Basico,
sem shift) ---
print("==============================================================")
print("Fase 1: Reducao para Hessenberg (via Householder)")
print("==============================================================")

```

```
from scipy.linalg import hessenberg
A1 = hessenberg(A)
# Set elements close to zero to zero for better visualization
A1[np.abs(A1) < 1e-12] = 0.0

print("Matriz A na forma de Hessenberg:")
print(A1)
print("--------------------------------------------------------------\

print("--------------------------------------------------------------\


# --- Fase 2: Metodo QR com Shift de Wilkinson (com
Deflacao) ---
print("=============================================================="
print("Fase 2: Metodo QR com Shift de Wilkinson e Deflacao")
print("O algoritmo foca na submatriz ativa (n x n),")
print("mas a matriz 5x5 inteira e exibida a cada passo.")
print("=============================================================="

A_k = np.copy(A1)
dim_total = 5
n = dim_total # 'n' e o tamanho da submatriz ativa
max_total_iter = 50 # Limite total de iteracoes
iter_count = 0
tolerance = 1e-12
tabela_convergencia = [] # <-- DADOS DA TABELA SERAO
ALTERADOS

# Autovalores verdadeiros ordenados para calculo do erro
autovalores_ordenados = np.sort(autovalores_desejados)

while n > 1 and iter_count < max_total_iter:
iter_count += 1

print(f"--- Fase 2: Iteracao {iter_count} (Foco na submatriz
{n}x{n}) ---")

# 1. Obter a submatriz ativa (para calculo do shift)
a = A_k[n-2, n-2]
b = A_k[n-1, n-1]
c = A_k[n-1, n-2]
```

```
76
77      # 2. Calcular o Shift de Wilkinson (mu)
78      delta = (a - b) / 2
79      if delta == 0:
80      mu = b - abs(c)
81      else:
82      sinal = np.sign(delta)
83      mu = b - (c**2) / (delta + sinal * np.sqrt(delta**2 + c**2))
84
85      print(f"(Shift mu = {mu:.6e})")
86
87      # 3. Aplicar o shift e QR na submatriz ativa A_k[0:n, 0:n]
88      I_sub = np.eye(n)
89      A_sub = A_k[0:n, 0:n]
90
91      A_shifted = A_sub - mu * I_sub
92      Q_k, R_k = np.linalg.qr(A_shifted)
93      A_sub = R_k @ Q_k + mu * I_sub
94
95      # 4. Reinserir a submatriz na matriz completa A_k
96      A_k[0:n, 0:n] = A_sub
97      # Set elements outside the three main diagonals to zero for
        the active submatrix
98      mask_tridiagonal = np.abs(np.arange(n)[:, None] -
        np.arange(n)[None, :]) > 1
99      A_k[0:n, 0:n][mask_tridiagonal] = 0.0
100
101     # 5. Imprimir a matriz 5x5 COMPLETA
102     print(A_k)
103
104     # 6. Coletar dados para a tabela
105     lambda_verdadeiro = autovalores_ordenados[n-1]
106     lambda_k = A_k[n-1, n-1]
107     erro = np.abs(lambda_k - lambda_verdadeiro)
108     m_val = np.abs(A_k[n-1, n-2])
109
110     # <-- ALTERADO: Armazenando (it, mu, erro, n) -->
111     # O valor de m (sub-diagonal) nao e mais armazenado na
        tabela.
112     tabela_convergencia.append((iter_count, mu, erro, n))
113
114     # 7. Checar Convergencia (Deflacao)
```

```
115     if m_val < tolerance:
116     print(f"\n*** CONVERGENCIA ALCANCADA PARA n={n} ***")
117     print(f"Autovalor encontrado: {lambda_k:.6e}")
118     print(f"Elemento sub-diagonal m = {m_val:.6e} (menor que a
        tolerancia)")
119
120     A_k[n-1, 0:n-1] = 0.0
121     A_k[0:n-1, n-1] = 0.0
122
123     n = n - 1
124     print(f"*** Reduzindo foco para a submatriz {n}x{n} ***\n")
125
126     # O ultimo autovalor e o que sobra em A_k[0,0]
127     print(f"\n*** CONVERGENCIA FINAL (n=1) ***")
128     print(f"Autovalor encontrado: {A_k[0,0]:.6e}")
129     print("Matriz Diagonalizada Final (A_k):")
130     print(A_k)
131     print("\nAutovalores encontrados na diagonal:")
132     print(np.diag(A_k))
133     print("\nAutovalores desejados (para comparacao):")
134     print(autovalores_desejados)
135
136     print("\n-------------------------------------------------------------
137
138
139     # --- Tabela de Convergencia (Fase 2) ---
140     print("==============================================================="
141     print("Tabela de Convergencia (Fase 2 - Wilkinson com
        Deflacao)")
142     # <-- ALTERADO: Titulo da tabela -->
143     print("lambda = autovalor alvo | Valor de m = Dimensao da
        submatriz ativa")
144     print("==============================================================="
145     # <-- ALTERADO: Cabecalho da tabela (removido 'n', 'Valor de
        m' agora e a dimensao) -->
146     print(" Iteracao |     mu (shift)     | |lambda_k - lambda|
        (Erro)  | Valor de m (Dimensao)")
147     print("----------|-------------------|-------------------|------------
148
149     # <-- ALTERADO: Loop de impressao da tabela -->
150     for dados in tabela_convergencia:
151     it, mu, erro, n_val = dados
```

```
152     # Imprime a dimensao 'n_val' (como inteiro) na ultima coluna
153     print(f" {it:8} | {mu:17.6e} | {erro:17.6e} | {n_val:21}")
154
155     # <-- ALTERADO: Texto explicativo -->
156     print("\nObserve como o 'Valor de m (Dimensao)' diminui,
        indicando a deflacao")
157     print("(reducao da matriz) a cada autovalor encontrado.")
```