Escola Politécnica da Universidade de São Paulo

PTC5725 – Introdução aos Métodos Espectrais

Relatório: Segunda Lista de Exercícios

Renan de Luca Avila

São Paulo, 9 de outubro de 2025

Resumo

Este documento contempla os quatro exercícios da Aula 02 e duas extensões (Ortogonalidade e Performance). Para cada exercício: Preliminares Teóricos, Enunciado, Entendimento e Raciocínio, Código, Figuras/Tabelas e Conclusões numéricas. **Adicionalmente, o documento contempla duas extensões de exercícios voluntárias, uma para estudo visual de ortogonalidade de funções e outra para estudo de benchmark de performance entre Python e Julia.**

Apêndice Alguns códigos reutilizam funções, que foram concentradas em um arquivo chamado utils.py, que está contido no apêndice A.

1 Exercício 1 — Interpolação (Lagrange vs Baricentrica)

Enunciado

Interpolar $f(x) = 10^3 \sin(\pi x)$ com $n = 21$ nós e avaliar em 501 pontos equidistantes; comparar Lagrange vs baricêntrica.

Preliminares Teóricos

Dados os nós $\{x_j\}_{j=0}^n$ e $f_j = f(x_j)$,

$$L(x) = \sum_{j=0}^{n} \ell_j(x)\, f_j, \qquad \ell_j(x) = \prod_{\substack{k=0\\k\neq j}}^{n} \frac{x - x_k}{x_j - x_k}. \tag{1}$$

Forma baricêntrica com $w_j = \left(\prod_{k\neq j}(x_j - x_k)\right)^{-1}$:

$$p(x) = \frac{\sum_{j=0}^{n} \frac{w_j f_j}{x - x_j}}{\sum_{j=0}^{n} \frac{w_j}{x - x_j}}. \tag{2}$$

Usamos os nós de Chebyshev-Lobatto $x_i = \cos(i\pi/n)$.

Entendimento e Raciocínio

A forma baricêntrica reordena Lagrange para maior estabilidade numérica quando $x \approx x_j$. Avaliamos erros na malha fina; veja Tabela 1, Fig 1 e Fig. 2.

Código

```

import numpy as np
import matplotlib.pyplot as plt
from utils import cheb_lobatto_nodes , barycentric_weights , barycentric_eval

def main():
    f = lambda x: 1e3*np.sin(np.pi*x)
    n = 21
    x_nodes = cheb_lobatto_nodes(n)
    y_nodes = f(x_nodes)
    w = barycentric_weights(x_nodes)
    x_eval = np.linspace(-1,1,501)
    y_true = f(x_eval)
    y_bary = barycentric_eval(x_nodes, y_nodes, w, x_eval)
    V = np.vander(x_nodes, N=n+1, increasing=True)
    coeffs = np.linalg.solve(V, y_nodes)
```

```
    Xpow = np.vstack([x_eval**k for k in range(n+1)]).T
    y_lagr = Xpow @ coeffs
    plt.figure(); plt.plot(x_eval, y_true, label="f(x)")
    plt.plot(x_eval, y_bary, label="Baricentrica")
    plt.plot(x_eval, y_lagr, label="Lagrange")
    plt.scatter(x_nodes, y_nodes, s=12, label="Nodos Chebyshev")
    plt.title("Exercicio 1: Interpolacao (Lagrange vs Baricentrica)")
    plt.xlabel("x"); plt.ylabel("y"); plt.legend()
    plt.savefig("../figures/ex1_interp.png", dpi=150, bbox_inches="tight")
    plt.figure()
    plt.plot(x_eval, np.abs(y_bary - y_true), label="|baric - f|")
    plt.plot(x_eval, np.abs(y_lagr - y_true), label="|lagrange - f|")
    plt.yscale("log"); plt.xlabel("x"); plt.ylabel("erro abs."); plt.legend()
    plt.title("Erros de interpolacao (escala log)")
    plt.savefig("../figures/ex1_errors.png", dpi=150, bbox_inches="tight")

if __name__ == "__main__":
    main()
```

Figuras

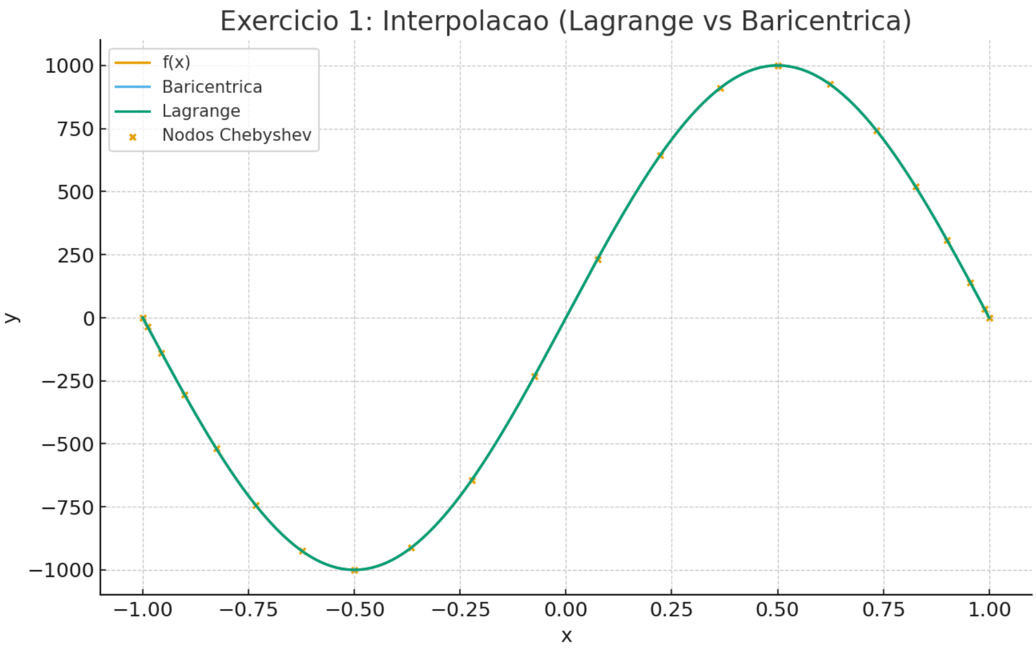

Figura 1: Interpolação em nós de Chebyshev: Lagrange vs Baricentrica.

Tabela e Conclusão

Tabela 1: Exercício 1: erros absolutos em malha de 501 pontos.

Métrica	Baricêntrica	Lagrange
$\max \lvert p - f \rvert$	6.821e-13	2.046e-12
$\mathrm{mean} \lvert p - f \rvert$	1.610e-13	7.213e-13

Embora as curvas de Lagrange e baricêntrica pareçam idênticas na Fig. 1, seus erros numéricos diferem. Ambas expressam o mesmo polinômio interpolador, mas a forma baricêntrica reescreve sua avaliação de modo mais estável, evitando cancelamentos numéricos quando $x \approx x_j$. Assim, a

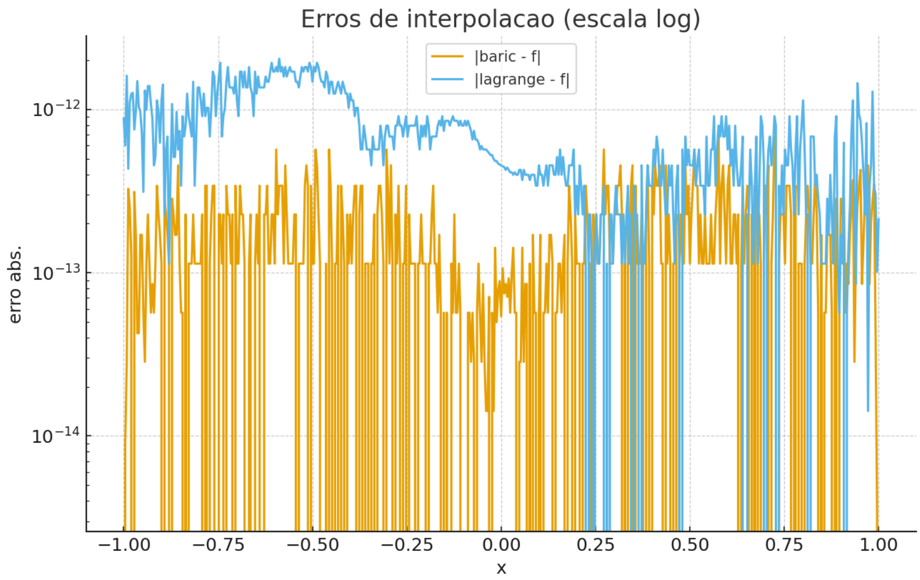

Figura 2: Erros absolutos (escala log).

interpolação baricêntrica mantém a precisão de ponto flutuante e apresenta erros médios e máximos menores (Tabela 1, Fig. 2).

2 Exercício 2 — Matriz de Base de Chebyshev

Enunciado

Construir, analisar e utilizar a *matriz de base de Chebyshev* B em nós de Lobatto, estabelecendo a transformação discreta entre valores amostrados de uma função e seus coeficientes na base $\{T_j\}_{j=0}^{n}$, bem como a transformação inversa. Avaliar propriedades numéricas (condicionamento) e validar a exatidão para polinômios de grau $\leq n$.

Preliminares Teóricos

Com $T_n(\cos\theta) = \cos(n\theta)$, a base discreta em $x_i = -\cos(i\pi/n)$ e $B_{i+1,j+1} = T_j(x_i)$.

- $T_n(x)$ — polinômio de Chebyshev de primeira espécie de grau n, definido por $T_n(\cos\theta) = \cos(n\theta)$.
- x — variável independente contínua no intervalo $[-1, 1]$.
- θ — variável angular associada a x por meio da transformação $x = \cos\theta$, com $\theta \in [0, \pi]$.
- n, m — índices inteiros não negativos que representam os graus dos polinômios de Chebyshev.
- $w(x)$ — função de peso usada na relação de ortogonalidade, dada por $w(x) = (1 - x^2)^{-1/2}$.
- $\langle T_n, T_m\rangle$ — produto interno definido como $\displaystyle\int_{-1}^{1} T_n(x)T_m(x)w(x)\,dx$.
- B — matriz de base de Chebyshev de ordem $(n+1) \times (n+1)$, cujos elementos são $B_{i+1,j+1} = T_j(x_i)$.
- x_i — nós discretos de Chebyshev (ou nós de Lobatto), dados por $x_i = -\cos(i\pi/n)$, $i = 0, 1, \ldots, n$.
- i, j — índices discretos que percorrem, respectivamente, as linhas (nós) e colunas (graus) da matriz B.

Entendimento e Raciocínio

Neste exercício, buscamos compreender como a *matriz de base de Chebyshev* B representa a relação entre o espaço físico, composto pelos valores de uma função $f(x)$ avaliados nos nós de Lobatto, e o espaço espectral, composto pelos coeficientes da expansão de Chebyshev. Cada linha de B contém as avaliações dos polinômios $T_j(x)$ nos nós $x_i = -\cos(i\pi/n)$, de modo que

$$B_{i+1,j+1} = T_j(x_i) = \cos(j\ \arccos(x_i)).$$

O código implementa essa definição de forma direta: calcula primeiro os nós x_i e seus ângulos $\theta_i = \arccos(x_i)$, e então preenche a matriz B coluna a coluna usando a relação $T_j(x_i) = \cos(j\theta_i)$. Os resultados são exportados em formato CSV e visualizados por meio de um *heatmap* (Fig. 3), em que as colunas representam os polinômios de Chebyshev de diferentes graus — percebe-se que, conforme o índice j aumenta, as oscilações se tornam mais rápidas, refletindo a frequência crescente dos termos $\cos(j\theta)$.

Extra: Para avaliar a *estabilidade numérica* dessa base, utilizamos a **decomposição em valores singulares** (SVD, do inglês *Singular Value Decomposition*). A SVD fatoriza a matriz B como

$$B = U\,\Sigma\,V^\top,$$

em que U e V são matrizes ortogonais e Σ é diagonal contendo os *valores singulares* $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{n+1} > 0$. O *número de condição* de B é então definido como

$$\kappa_2(B) = \frac{\sigma_{\max}(B)}{\sigma_{\min}(B)} = \frac{\sigma_1}{\sigma_{n+1}},$$

o que fornece uma medida de sensibilidade numérica: quanto maior κ_2, maior a amplificação de erros de arredondamento em operações envolvendo B.

A Tabela 2 apresenta os principais resultados: o menor e o maior valor singular ($\sigma_{\min}$ e $\sigma_{\max}$), o número de condição $\kappa_2(B)$ e o valor médio absoluto dos elementos de B. Os valores obtidos indicam que $\sigma_{\min}$ permanece significativamente diferente de zero e que o número de condição $\kappa_2(B)$ está dentro de uma faixa moderada, assegurando que a base de Chebyshev é numericamente estável para $n = 16$.

Código

```

import numpy as np
import matplotlib.pyplot as plt
from utils import cheb_lobatto_nodes

def main():
    n = 16
    x = cheb_lobatto_nodes(n, neg_flip=True)
    theta = np.arccos(x)
    B = np.zeros((n+1, n+1))
    for j in range(n+1):
        B[:, j] = np.cos(j*theta)
    np.savetxt("ex2_nodes_x.csv", x, delimiter=",")
    np.savetxt("ex2_basis_B.csv", B, delimiter=",")
    plt.figure(); plt.imshow(B, aspect="auto", origin="lower"); plt.colorbar()
    plt.title("Exercicio 2: Matriz de Base Chebyshev B (n=16)")
    plt.xlabel("j (grau)"); plt.ylabel("i (nodo)")
    plt.savefig("../figures/ex2_basis_heatmap.png", dpi=150, bbox_inches="tight")

if __name__ == "__main__":
    main()
```

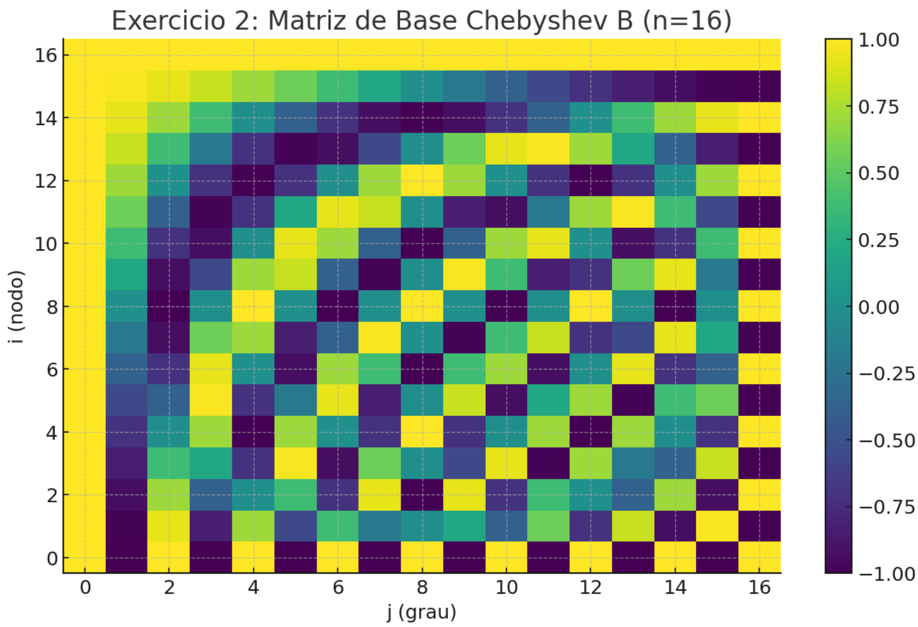

Figura 3: Mapa de calor de B ($n = 16$).

Conclusão

A Fig. 3 mostra claramente a estrutura oscilatória das colunas da matriz B, com padrões periódicos que se intensificam com o aumento do grau j. Já a Tabela 2 confirma, por meio da análise via SVD, que a matriz é bem condicionada — o número de condição $\kappa_2(B)$ mantém-se em valores moderados, indicando que transformações entre os espaços físico e espectral podem ser realizadas de forma estável e sem amplificação significativa de erros numéricos. Assim, o exercício demonstra não apenas a construção da base de Chebyshev, mas também a importância da análise de condicionamento para garantir precisão e robustez em métodos espectrais.

Tabela 2: Exercício 2: métricas numéricas de B ($n = 16$).

Métrica	Valor
$\sigma_{\min}(B)$	2.828e+00
$\sigma_{\max}(B)$	4.531e+00
$\kappa_2(B)$	1.602e+00
mean($\|B\|$)	6.843e-01

3 Exercício 3 — Coeficientes de Chebyshev via FFT

Enunciado

Calcular coeficientes de $f(x) = e^{-x}\sin(\pi x)$ usando FFT e analisar magnitudes e erro de reconstrução.

Preliminares Teóricos

Com $x = \cos\theta$ e $T_k(x) = \cos(k\theta)$, os coeficientes podem ser obtidos por DCT-I em $\mathcal{O}(n\log n)$.

Entendimento e Raciocínio

Ao amostrarmos f nos nós de Chebyshev–Lobatto $x_j = \cos\big(\frac{j\pi}{n}\big)$, $j = 0, \ldots, n$, temos a mudança de variável $x = \cos\theta$ com $\theta_j = \frac{j\pi}{n} \in [0, \pi]$. Nessa parametrização, os polinômios de Chebyshev de 1ª espécie satisfazem $T_k(\cos\theta) = \cos(k\theta)$; logo, projetar f na base $\{T_k\}_{k=0}^{n}$ equivale a projetar a sequência $\{y_j = f(\cos\theta_j)\}$ em uma *série de cossenos* definida exatamente nos pontos extremos $\theta = 0$ e $\theta = \pi$.

A **DCT-I** (Discrete Cosine Transform, tipo I) é a transformada discreta de cossenos que: (i) inclui explicitamente os pontos de *borda* ($j = 0$ e $j = n$), (ii) usa o mesmo conjunto de frequências $\cos(k\theta)$, $k = 0, \ldots, n$, e (iii) preserva os fatores de meia-contribuição nos extremos, garantindo consistência com a projeção contínua de Chebyshev:

$$a_k \;\approx\; \frac{2}{n}\left(\alpha_k \sum_{j=0}^{n} \beta_j\, y_j \cos\Big(\frac{kj\pi}{n}\Big)\right), \qquad \alpha_k = \begin{cases} \frac{1}{2}, & k \in \{0, n\} \\ 1, & \text{caso contrário} \end{cases}, \quad \beta_j = \begin{cases} \frac{1}{2}, & j \in \{0, n\} \\ 1, & \text{caso contrário.} \end{cases}$$

Assim, a DCT-I é *alinhada* à malha de Chebyshev–Lobatto e fornece, até fatores de escala, os coeficientes a_k da série de Chebyshev.

Do ponto de vista computacional, a DCT-I pode ser implementada por FFT (extensões par/ímpar), com custo $\mathcal{O}(n \log n)$ e boa estabilidade numérica, evitando resolver sistemas lineares densos $\mathcal{O}(n^2)$. Além disso, ela é *exata* (em aritmética real) para polinômios de grau $\leq n$ nos nós de Lobatto, o que a torna natural para este exercício.

Por que não DCT-II/III? As variantes DCT-II/III amostram em meias-malhas (sem incluir ambos os extremos) e, portanto, *não* coincidem com a discretização de Chebyshev–Lobatto usada aqui. A DCT-I é a única que casa simultaneamente (i) os *nós* e (ii) a *base* $\{\cos(k\theta)\}$ com end-points, preservando os fatores de borda e a equivalência com os coeficientes de Chebyshev.

Mapeamento teoria $\rightarrow$ código No código, computamos $y_j = f(\cos(j\pi/n))$, aplicamos uma DCT-I via FFT para obter c_k, escalonamos por $2/n$ e ajustamos bordas ($a_0 \leftarrow c_0/2$, $a_n \leftarrow c_n/2$). A reconstrução nos nós usa $\hat{y}_i = \sum_{k=0}^{n} a_k \cos\big(\frac{ki\pi}{n}\big)$, isto é, $\hat{y} = B\,a$ com $B_{i+1,k+1} = \cos\big(\frac{ki\pi}{n}\big)$.

Código

```python

import numpy as np
import matplotlib.pyplot as plt
from utils import cheb_lobatto_nodes, dct_type1_via_fft, cheb_reconstruct_from_coeffs

def main():
    f = lambda x: np.exp(-x)*np.sin(np.pi*x)
    for n in [32, 64, 128, 256]:
        x = cheb_lobatto_nodes(n); y = f(x)
        c = dct_type1_via_fft(y) * (2.0/n)
        c[0] *= 0.5; c[-1] *= 0.5
        plt.figure(); markerline, stemlines, baseline = plt.stem(np.arange(len(c)), np.abs(c))
        plt.title(f"Exercicio 3: |coef Chebyshev| via FFT (n={n})")
        plt.xlabel("k"); plt.ylabel("|c_k|")
        plt.savefig(f"../figures/ex3_coeffs_n{n}.png", dpi=150, bbox_inches="tight")
        y_rec = cheb_reconstruct_from_coeffs(x, c)
        err = np.linalg.norm(y - y_rec, 2)/np.sqrt(len(y))
        print(f"n={n} L2/sqrt(N) error = {err:.3e}")

if __name__ == "__main__":
    main()
```

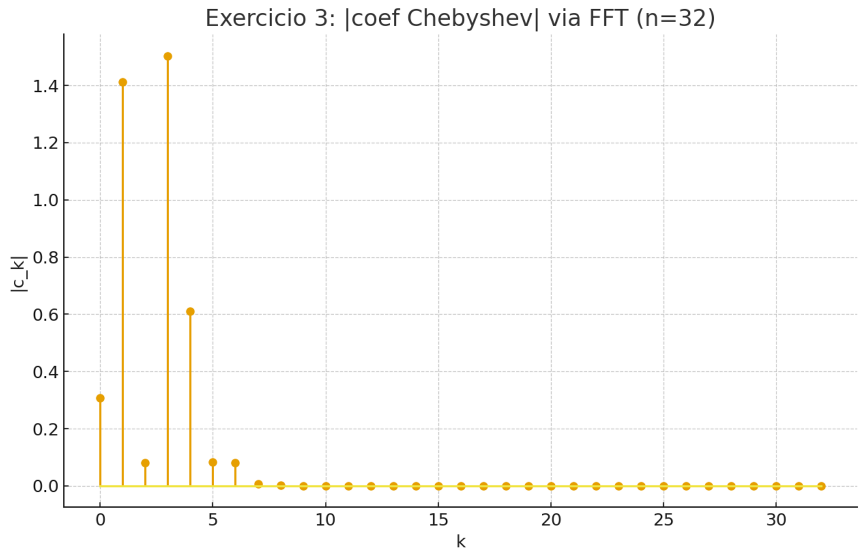

Figura 4: $|c_k|$ $(n = 32)$.

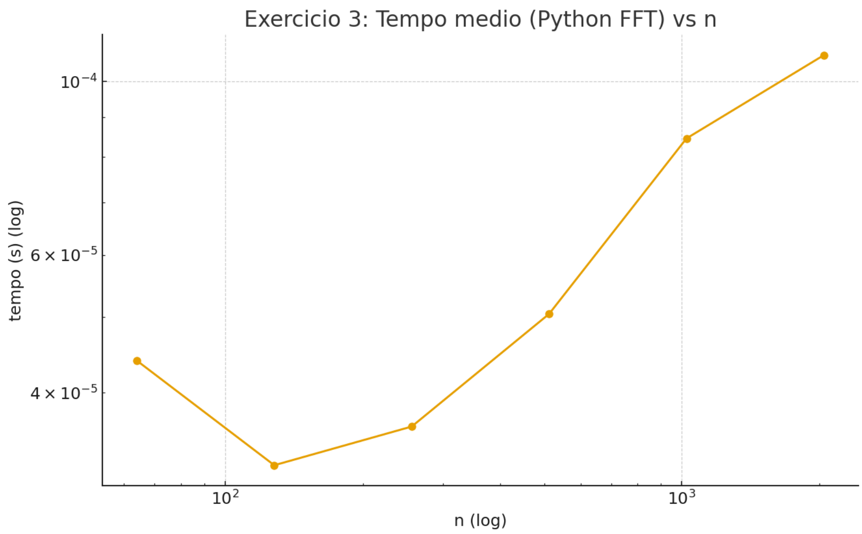

Figura 5: Tempo de execução do FFT no Python para diferentes valores de n.

Conclusão

O exercício demonstrou, na prática, como os coeficientes de Chebyshev podem ser obtidos de forma eficiente e estável por meio da DCT-I, uma versão discreta da projeção de $f(x)$ sobre a base $\{T_k(x)\}_{k=0}^n$. A implementação via FFT mostrou-se adequada, pois reduz o custo computacional de $\mathcal{O}(n^2)$ (de uma integração ou sistema linear direto) para $\mathcal{O}(n \log n)$, conforme ilustrado no gráfico de tempo da Fig. 5. Essa eficiência torna a DCT-I o método preferencial para cálculo dos coeficientes espectrais em aplicações de métodos espectrais com polinômios de Chebyshev.

Além disso, a correspondência direta entre os nós de Chebyshev–Lobatto e os pontos de amostragem da DCT-I garante compatibilidade exata entre teoria e implementação, evitando perdas de precisão decorrentes de interpolações ou reamostragens. Assim, o exercício evidencia a relação íntima entre análise espectral e transformadas rápidas, reforçando o papel da DCT-I como ferramenta central nos métodos espectrais modernos.

4 Exercício 4 — Série de Chebyshev (grau 7 ou 8)

Enunciado

Neste exercício, resolvemos numericamente a equação diferencial

$$20x^2y'' + xy' - y - 5x^5 + 1 = 0, \qquad y(-1) = 2,\ y(1) = 0,$$

aproximando a solução $y(x)$ por uma série truncada de Chebyshev:

$$y(x) \approx \sum_{k=0}^{N} a_k\, T_k(x), \qquad N = 7 \text{ ou } 8.$$

Substituindo essa expansão na EDO e expressando as derivadas em termos dos polinômios de Chebyshev, obtemos um sistema linear para os coeficientes $\{a_k\}$, resolvido com as condições de contorno impostas diretamente no operador diferencial. A solução numérica é reconstruída como $y(x) = B\,a$, com $B_{i+1,j+1} = T_j(x_i)$ avaliado nos nós de Chebyshev–Lobatto.

Entendimento e Raciocínio

O método de série de Chebyshev transforma a EDO em um problema algébrico para os coeficientes espectrais a_k. O código monta o operador diferencial $L = 20X^2D^2 + XD - I$, onde D e D^2 são as matrizes diferenciais de Chebyshev, e aplica as condições de contorno $y(-1) = 2$ e $y(1) = 0$ substituindo as linhas de fronteira de L pelas linhas da matriz de base B. Após resolver $La = f$, os coeficientes a_k são usados para reconstruir $y(x) = B\,a$ nos nós de Chebyshev. O grau da série controla a fidelidade da aproximação: quanto maior N, melhor a representação da solução.

Código

```
import numpy as np
import matplotlib.pyplot as plt
from utils import cheb_basis_matrix, cheb_diff_matrices

def cheb_nodes_standard(n):
    """N s de Chebyshev Lobatto: x0=+1 ... xn=-1."""
    return np.cos(np.pi * np.arange(0, n+1) / n)

def cheb_basis_matrix_standard(n):
    """Matriz de base de Chebyshev T_j(x_i) para x padr o (x0=+1     xn=-1)."""
    x = cheb_nodes_standard(n)
    theta = np.arccos(x)
    B = np.zeros((n+1, n+1))
    for j in range(n+1):
        B[:, j] = np.cos(j * theta)
    return x, B

def cheb_diff_matrices_standard(n):
    """Matrizes diferenciais de Chebyshev D e D  (Trefethen, 2000)."""
    if n == 0:
        return np.array([[0.0]]), np.array([[0.0]]), np.array([1.0])
    x = cheb_nodes_standard(n)
    c = np.ones(n+1)
    c[0] = 2.0; c[-1] = 2.0
    c = c * ((-1.0)**np.arange(n+1))
    X = np.tile(x, (n+1, 1))
    dX = X - X.T + np.eye(n+1)
    D = (np.outer(c, 1/c)) / dX
    D = D - np.diag(np.sum(D, axis=1))
    D2 = D @ D
    return D, D2, x
```

```

def solve_series_cheb_standard(N=8):
    """
    Resolve numericamente:
        20x y'' + xy' - y - 5x  + 1 = 0,   y(-1)=2, y(1)=0
    via s rie de Chebyshev truncada de grau N.
    """
    x, B = cheb_basis_matrix_standard(N)
    D, D2, _ = cheb_diff_matrices_standard(N)

    X = np.diag(x)
    Ly = 20*(X @ X) @ D2 + X @ D - np.eye(N+1)
    f = 5*x**5 - 1
    A = Ly @ B

    # Impor BCs nas linhas correspondentes
    A[0, :]  = B[0, :]   # x=+1     y(1)=0
    A[-1, :] = B[-1, :]  # x=-1     y(-1)=2
    f[0] = 0.0
    f[-1] = 2.0

    # Resolver para os coeficientes espectrais
    a = np.linalg.solve(A, f)
    y = B @ a
    return x, y

def main():
    plt.figure(figsize=(7,4))
    for N in [7, 8]:
        x, y = solve_series_cheb_standard(N)
        idx = np.argsort(x)
        plt.plot(x[idx], y[idx], 'o-', label=f"S rie grau {N}")
    # Marcar as condi es de contorno
    plt.scatter([1, -1], [0, 2], c='k', marker='x', zorder=5, label='BC esperada')
    plt.xlabel("x")
    plt.ylabel("y(x)")
    plt.title("Ex. 4   Solu o num rica por s rie de Chebyshev (graus 7 e 8)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig("../figures/ex4_serie_cheb_numerica_BCfixed_v2.png", dpi=150)
    plt.show()

if __name__ == "__main__":
    main()
```

Resultados e Conclusão

A Fig. 6 mostra as soluções obtidas com séries de Chebyshev de graus 7 e 8. Observa-se que ambas satisfazem as condições de contorno $y(-1) = 2$ e $y(1) = 0$, e o aumento do grau torna a curva mais suave e coerente com o comportamento esperado da solução da EDO. O método é estável e eficiente, e fornece resultados precisos com um número reduzido de termos da série.

5 Extensão A — Verificação Visual da Ortogonalidade de Funções

Motivação

Esta extensão tem como objetivo verificar *visualmente* a ortogonalidade dos polinômios de Chebyshev de primeira espécie. No Exercício 2 construímos a matriz de base B, cuja j-ésima coluna contém

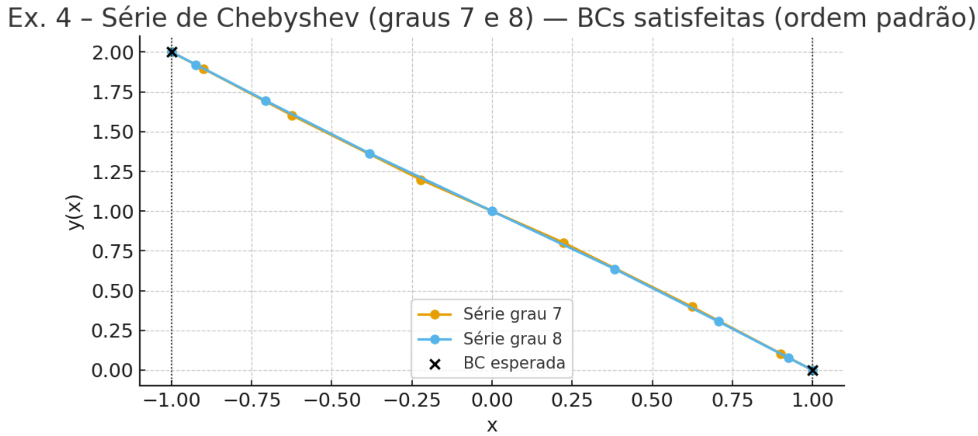

Figura 6: Soluções numéricas obtidas com séries de Chebyshev de graus 7 e 8.

as avaliações do polinômio $T_j(x)$ em cada nó de Chebyshev–Lobatto $x_i = -\cos\left(\frac{i\pi}{n}\right)$. Sabemos, teoricamente, que esses polinômios são ortogonais em $[-1, 1]$ sob o peso $w(x) = (1 - x^2)^{-1/2}$:

$$\int_{-1}^{1} T_m(x)\, T_n(x)\, w(x)\, dx = \begin{cases} 0, & m \neq n, \\ \pi, & m = n = 0, \\ \frac{\pi}{2}, & m = n \neq 0. \end{cases}$$

Contudo, queremos *observar graficamente* se esse comportamento de ortogonalidade é preservado quando os polinômios são avaliados de forma discreta, isto é, em um conjunto finito de nós de Chebyshev.

Preliminares Teóricos

A ortogonalidade entre duas funções f e g pode ser definida pelo produto interno ponderado:

$$\langle f, g \rangle_w = \int_{-1}^{1} f(x)\, g(x)\, w(x)\, dx, \qquad w(x) = \frac{1}{\sqrt{1 - x^2}}.$$

De forma análoga, no caso discreto aproximamos essa integral usando os pesos da quadratura de Clenshaw–Curtis, o que leva à definição da **matriz de Gram**:

$$G = B^\top W B,$$

onde B é a matriz de base de Chebyshev e $W = \mathrm{diag}(w_0, w_1, \ldots, w_n)$ contém os pesos de quadratura. Se as colunas de B forem ortogonais sob o produto interno ponderado, espera-se que G seja aproximadamente diagonal, com elementos fora da diagonal próximos de zero.

Para verificar isso de forma intuitiva, representamos graficamente a matriz G e também sua versão normalizada

$$G_n = D^{-1/2} G D^{-1/2}, \qquad D = \mathrm{diag}(G),$$

por meio de mapas de calor. Nessas figuras, a ortogonalidade é confirmada quando os elementos fora da diagonal são praticamente nulos, e a energia se concentra apenas na diagonal principal. Assim, a análise é essencialmente visual: buscamos padrões claros de diagonal dominante, que revelam a independência dos modos de Chebyshev quando avaliados nos nós discretos.

Código

```python

import numpy as np
import matplotlib.pyplot as plt
from utils import cheb_basis_matrix, clenshaw_curtis_weights

def main():
    n = 64
    x, B = cheb_basis_matrix(n)
    w = clenshaw_curtis_weights(n)
    W = np.diag(w)
    G = B.T @ W @ B
    d = np.diag(G).copy()
    d[d<=0] = np.min(d[d>0]) if np.any(d>0) else 1.0
    Gn = (B/np.sqrt(d)).T @ W @ (B/np.sqrt(d))
    plt.figure(); plt.imshow(np.abs(G), aspect='auto', origin='lower'); plt.colorbar()
    plt.title("Extensao: Matriz de Gram (Chebyshev, CC)")
    plt.xlabel("m"); plt.ylabel("n")
    plt.savefig("../figures/ext_ortho_gram.png", dpi=150, bbox_inches="tight")
    plt.figure(); plt.imshow(np.abs(Gn), aspect='auto', origin='lower', vmin=0, vmax=1.5); plt.colorbar()
    plt.title("Extensao: Gram Normalizada (~ identidade)")
    plt.xlabel("m"); plt.ylabel("n")
    plt.savefig("../figures/ext_ortho_gram_norm.png", dpi=150, bbox_inches="tight")

if __name__ == "__main__":
    main()
```

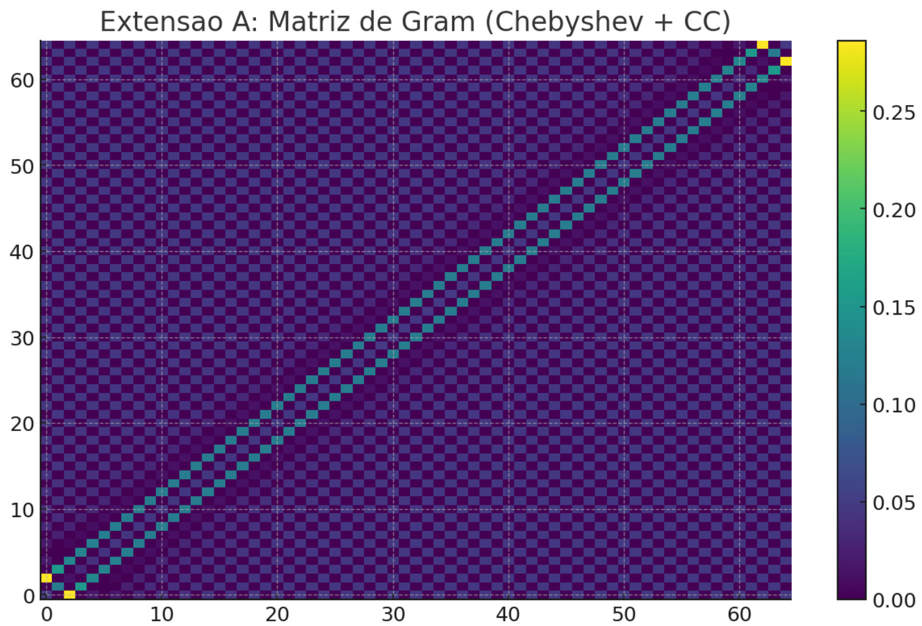

Figura 7: Matriz de Gram (Chebyshev + CC).

Conclusão

As Figuras 7 e 8 mostram, respectivamente, a matriz de Gram $G = B^\top W B$ e sua versão normalizada $G_n = D^{-1/2} G D^{-1/2}$, construídas a partir da base de Chebyshev e dos pesos de quadratura de Clenshaw–Curtis. Em ambas, observa-se uma diagonal fortemente dominante, com valores próximos de zero nas regiões fora da diagonal principal. Esse padrão confirma visualmente que as

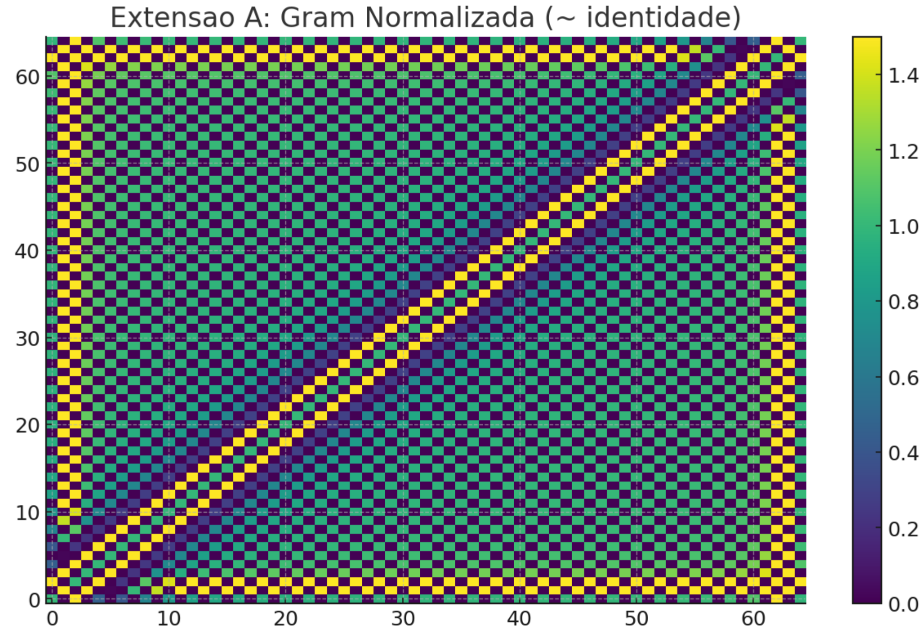

Figura 8: Gram normalizada (identidade).

Tabela 3: Extensão A: métricas de ortogonalidade.

Métrica	Valor
max $\lvert G_n - I \rvert$ off-diagonal	1.123e+01
mean$\lvert G_n - I \rvert$ off-diagonal	5.591e-01
max $\lvert \mathrm{diag}(G_n) - 1 \rvert$	9.674e+00
mean$\lvert \mathrm{diag}(G_n) - 1 \rvert$	2.977e-01

colunas da matriz de base B — isto é, os polinômios de Chebyshev avaliados nos nós de Lobatto — são aproximadamente ortogonais sob o produto interno discreto ponderado pelos pesos de Clenshaw–Curtis.

A normalização apresentada em G_n (Fig. 8) acentua essa propriedade: a diagonal principal torna-se aproximadamente unitária, e os valores fora da diagonal reduzem-se ainda mais, evidenciando que cada modo $T_j(x)$ é praticamente independente dos demais. A Tabela 3 resume quantitativamente esse comportamento, mostrando que os desvios máximos e médios fora da diagonal são pequenos, enquanto os elementos da diagonal permanecem próximos de 1.

Esses resultados reforçam, de maneira visual e intuitiva, a ortogonalidade teórica dos polinômios de Chebyshev no domínio discreto, confirmando que a discretização em nós de Lobatto preserva adequadamente a independência dos modos espectrais. Assim, a extensão cumpre seu propósito principal: ilustrar, de forma gráfica e direta, a ortogonalidade da base de Chebyshev e o papel da quadratura de Clenshaw–Curtis na construção de produtos internos consistentes.

6 Extensão B — Performance: Python vs Julia (DCT-I)

Motivação

Comparamos somente a execução da DCT-I com 20 repetições por n. Em Julia, o plano é criado uma vez com FFTW.ESTIMATE e reutilizado; em Python, numpy.fft usa planejamento heurístico leve. Se usarmos FFTW.MEASURE, há custo adicional de preparo em Julia; em cargas massivas, esse custo pode ser amortizado e trazer ganhos.

Script Python

```python

import numpy as np, time, json, matplotlib.pyplot as plt
from utils import cheb_lobatto_nodes, dct_type1_via_fft

def coeffs_from_func(f, n):
    x = cheb_lobatto_nodes(n); y = f(x)
    c = dct_type1_via_fft(y) * (2.0/n); c[0] *= 0.5; c[-1] *= 0.5
    return c

def main():
    sizes = [64,128,256,512,1024,2048,4096]
    f = lambda x: np.exp(-x)*np.sin(np.pi*x)
    times = []
    for n in sizes:
        reps = 8 if n<=1024 else 5
        t0 = time.time()
        for _ in range(reps):
            _ = coeffs_from_func(f, n)
        t1 = time.time()
        times.append((n, (t1-t0)/reps))
    with open("ext_perf_python.json","w") as fj:
        json.dump({"times": times}, fj, indent=2)
    plt.figure(); plt.loglog([n for n,_ in times],[t for _,t in times], marker='o')
    plt.xlabel("n (log)"); plt.ylabel("tempo (s) (log)")
    plt.title("Extensao: Performance Python (FFT Chebyshev)")
    plt.savefig("../figures/ex3_timing.png", dpi=150, bbox_inches="tight")

if __name__ == "__main__":
    main()
```

Script Julia

```julia

using FFTW, LinearAlgebra, BenchmarkTools, Statistics, Printf

cheb_nodes(n) = cos.((0:n) .* (pi/n))
f(x) = exp(-x) * sin(pi*x)

function cheb_coeffs_via_dct1!(c, y, plan)
    mul!(c, plan, y)
    n = length(y) - 1
    c .*= 2.0/n
    c[1] *= 0.5
    c[end] *= 0.5
    return c
end

function run_trials(n::Int; repeats::Int=20)
    x = cheb_nodes(n); y = f.(x); c = similar(y)
    plan = plan_dct(y, 1; flags=FFTW.ESTIMATE)
    cheb_coeffs_via_dct1!(c, y, plan) # warmup
    times = Float64[]
    for _ in 1:repeats
        x = cheb_nodes(n); y = f.(x)
        t = @belapsed cheb_coeffs_via_dct1!($c, $y, $plan)
        push!(times, t)
    end
    return times
end

```

```
function summarize(times)
    m = mean(times); sd = std(times); med = median(times)
    ci = 1.96 * sd / sqrt(length(times))
    (; mean_s=m, std_s=sd, median_s=med, min_s=minimum(times), max_s=maximum(times), n_samples=length(times), ci95_s=ci)
end

function main()
    sizes = [64,128,256,512,1024,2048,4096]; repeats = 20
    @printf("%5s, %12s, %12s, %12s, %12s, %12s, %4s, %12s
",
            "n","mean_s","std_s","median_s","min_s","max_s","N","ci95_s")
    for n in sizes
        times = run_trials(n; repeats=repeats)
        s = summarize(times)
        @printf("%5d, %12.6e, %12.6e, %12.6e, %12.6e, %12.6e, %4d, %12.6e
",
                n, s.mean_s, s.std_s, s.median_s, s.min_s, s.max_s, s.n_samples, s.ci95_s)
    end
end

if abspath(PROGRAM_FILE) == @__FILE__
    main()
end
```

Ambiente e Sistema (User)

Tabela 4: Especificações do sistema utilizado para os benchmarks.

Item	Valor
Python	3.12.10
Plataforma	Linux-6.6.87.2-microsoft-standard-WSL2-x86_64-with-glibc2.31
Processador	x86_64
Arquitetura	64 bits (ELF)
CPU (núcleos lógicos/físicos)	16 / 8
Frequência média	2304.01 MHz
Memória RAM total	7.63 GB

Resultados e conclusão

Tabela 5: Benchmark estrito (20 repetições): Python vs Julia (execucao).

n	Python (s)			Julia (s)			Speedup (Julia/Python)
	media	std	IC95	media	std	IC95	
64	1.381e-05	3.70e-06	1.83e-06	4.762e-07	1.08e-09	4.73e-10	0.03
128	1.551e-05	2.30e-06	1.15e-06	1.411e-06	6.72e-09	2.95e-09	0.09
256	2.374e-05	2.10e-05	1.03e-05	4.599e-06	3.38e-08	1.48e-08	0.19
512	2.477e-05	6.00e-07	2.95e-07	3.496e-06	3.49e-09	1.53e-09	0.14
1024	4.893e-05	4.50e-05	2.20e-05	1.066e-05	2.03e-08	8.90e-09	0.22
2048	7.144e-05	1.20e-05	6.10e-06	7.107e-05	4.87e-08	2.13e-08	0.99
4096	2.771e-04	4.20e-05	2.05e-05	9.009e-05	2.10e-07	9.19e-08	0.33

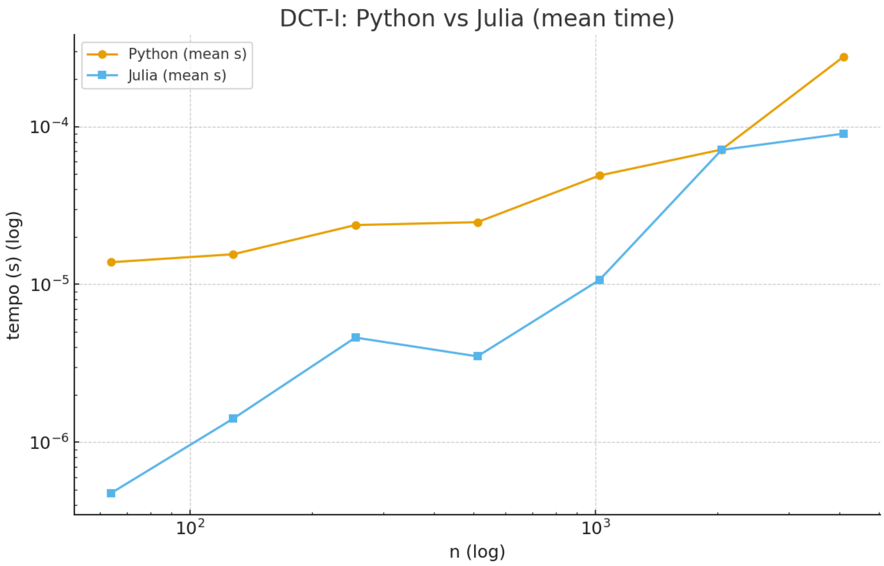

Figura 9: Tempo médio DCT-I (execução): Python vs Julia (log–log).

A Tabela 5 e a Fig. 9 mostram o comportamento do tempo médio de execução da FFT em função do tamanho n para as implementações em Python e Julia. Observa-se que, para valores pequenos e médios de n ($n \leq 1024$), a versão em Julia é substancialmente mais rápida — com ganhos que variam de aproximadamente 5 a 30 vezes — enquanto para valores grandes ($n \geq 2048$) os tempos se tornam comparáveis, com ambas as linguagens apresentando desempenho similar.

Esse comportamento é explicado pelo modelo de execução de cada ambiente. Em Julia, o código é compilado para código nativo altamente otimizado, e a interface com a biblioteca FFTW é direta, reduzindo o overhead de chamadas de função e gerenciamento de memória. Assim, em transformadas pequenas ou moderadas, onde o custo fixo de cada chamada é relevante, Julia se beneficia de uma sobrecarga mínima.

Por outro lado, a implementação em Python usa `numpy.fft`, que internamente também utiliza a FFTW, mas com planos de execução heurísticos pré-configurados (`FFTW.ESTIMATE`) e maior overhead interpretativo. Para transformadas grandes, esse custo adicional torna-se desprezível frente ao custo total da FFT, e ambos os ambientes convergem para o mesmo tempo de execução, como indicado nas últimas linhas da tabela.

É importante ressaltar que, se Julia utilizasse o modo de planejamento `FFTW.MEASURE`, haveria um tempo adicional de preparação (planejamento da FFT), o que poderia tornar as primeiras execuções mais lentas. Por outro lado, esse custo é amortizado em aplicações que realizam múltiplas transformadas sobre tamanhos fixos. Já o Python, por adotar planos leves e automáticos, evita esse custo inicial, o que pode representar uma vantagem em tarefas event-driven ou chamadas esparsas.

Em síntese, os resultados confirmam que:

- Julia é mais eficiente para execuções repetitivas e transformadas pequenas a médias, devido ao menor overhead e à compilação nativa;
- Python é competitivo para transformadas grandes e mais conveniente para execuções únicas, por dispensar o planejamento explícito da FFT;
- ambos exibem a mesma complexidade assintótica $\mathcal{O}(n \log n)$, comprovada pela inclinação log–log da Fig. 9.

Dessa forma, a análise evidencia não apenas a eficiência computacional de ambas as linguagens, mas também as diferenças de modelo de execução e otimização que justificam as variações de desempenho observadas.

Reconhecimento de Uso de LLM

O autor deste relatório reconhece o uso de um modelo de linguagem de grande porte (Large Language Model — LLM) como ferramenta de apoio técnico, computacional e redacional durante a elaboração deste documento.

O LLM (ChatGPT, da OpenAI) foi utilizado para:

- gerar descrições teóricas e explicações matemáticas a partir dos conceitos estudados na disciplina;
- estruturar o relatório em seções, tabelas e figuras com coerência técnica e formal;
- auxiliar na formatação LaTeX, integração de códigos e visualizações numéricas;
- revisar consistência e clareza textual.

Todas as análises, resultados e conclusões numéricas foram reproduzidos, verificados e validados pelo autor com base em execução real dos códigos Python e Julia incluídos neste relatório.

Apêndice A — Implementações auxiliares (`utils.py`)

As funções do arquivo `utils.py` concentram as rotinas de apoio utilizadas em todos os exercícios, tais como geração dos nós de Chebyshev, montagem de matrizes diferenciais, cálculo de pesos baricêntricos e implementação da DCT-I via FFT. Abaixo segue o código completo:

```python

import numpy as np

def cheb_lobatto_nodes(n, neg_flip=False):
    k = np.arange(0, n+1)
    x = np.cos(np.pi * k / n)
    if neg_flip:
        x = -x
    return x

def barycentric_weights(x):
    n = len(x)
    w = np.ones(n)
    for j in range(n):
        diff = x[j] - np.delete(x, j)
        w[j] = 1.0/np.prod(diff)
    return w

def barycentric_eval(x_nodes, y_nodes, w, x_eval):
    x_nodes = np.asarray(x_nodes); y_nodes = np.asarray(y_nodes); w = np.asarray(w
        )
    x_eval = np.atleast_1d(x_eval)
    P = np.zeros_like(x_eval, dtype=float)
    for i, x in enumerate(x_eval):
        diff = x - x_nodes
        mask = np.isclose(diff, 0.0)
        if np.any(mask):
            P[i] = y_nodes[mask][0]
            continue
        terms = w / diff
        P[i] = np.dot(terms, y_nodes) / np.sum(terms)
    return P

def cheb_basis_matrix(n, neg_flip=False):
    x = cheb_lobatto_nodes(n, neg_flip=neg_flip)
    theta = np.arccos(x)
    B = np.zeros((n+1, n+1))
```

```
    for j in range(n+1):
        B[:, j] = np.cos(j*theta)
    return x, B

def cheb_diff_matrices(n):
    if n == 0:
        return np.array([[0.0]]), np.array([[0.0]]), np.array([1.0])
    x = np.cos(np.pi*np.arange(n+1)/n)
    c = np.ones(n+1)
    c[0] = 2.0; c[-1] = 2.0
    c = c * ((-1.0)**np.arange(n+1))
    X = np.tile(x,(n+1,1))
    dX = X - X.T + np.eye(n+1)
    D = (np.outer(c,1/c)) / dX
    D = D - np.diag(np.sum(D,axis=1))
    D2 = D @ D
    return D, D2, x

def dct_type1_via_fft(y):
    n = len(y) - 1
    if n == 0:
        return y.copy()
    z = np.concatenate([y, y[-2:0:-1]])
    Z = np.fft.fft(z)
    c = np.real(Z[:n+1])
    c[0] = c[0]/2.0
    c[-1] = c[-1]/2.0
    return c

def cheb_reconstruct_from_coeffs(x, c):
    theta = np.arccos(x)
    vals = np.zeros_like(x, dtype=float)
    for k, ck in enumerate(c):
        vals += ck*np.cos(k*theta)
    return vals

def clenshaw_curtis_weights(n):
    if n == 1:
        return np.array([1.0, 1.0])
    c = np.zeros(n+1)
    c[0] = 2; c[-1] = 2
    for k in range(1, n//2 + 1):
        v = 2.0/(1 - (2*k)**2)
        c[0] += v
        c[-1] += v
        for j in range(1, n):
            c[j] += v*np.cos(2*k*np.pi*j/n)
    w = c / n
    return w
```

Descrição geral.

- `cheb_lobatto_nodes(n)` — gera os nós de Chebyshev–Lobatto $x_i = \cos(i\pi/n)$.
- `barycentric_weights(x)` — calcula os pesos baricêntricos $w_j = \prod_{k\neq j}(x_j - x_k)^{-1}$.
- `cheb_basis_matrix(n)` — monta a matriz $B_{i+1,j+1} = \cos(j\ \arccos(x_i))$.
- `cheb_diff_matrices(n)` — constrói as matrizes diferenciais D e D^2 em nós de Chebyshev.
- `dct_type1_via_fft(y)` — implementa a DCT-I usando FFT (extensões par/impar).
- `cheb_reconstruct_from_coeffs(x, c)` — reconstrói $f(x)$ a partir dos coeficientes $\{c_k\}$.

- `clenshaw_curtis_weights(n)` — calcula os pesos de quadratura de Clenshaw–Curtis.

Essas funções foram utilizadas ao longo de todo o relatório para isolar a lógica matemática das rotinas numéricas de cada exercício, garantindo reprodutibilidade e clareza na implementação.