Escola Politécnica da Universidade de São Paulo

# Aula 06 – Métodos Espectrais

Resolução da tarefa

Renan de Luca Avila
30 de outubro de 2025

# Sumário

# 1 Resumo

Este relatório implementa, em Python, os exercícios apresentados nos slides da Aula 06 (Prof. Osvaldo Guimarães), e discute as soluções. O documento contém códigos completos, figuras e tabelas, além de apêndices com setup e fontes.

# 2 Resoluções

Cada resolução é apresentada com planejamento, resultados, conclusão e implementação. Os códigos completos estão no Apêndice **??**.

## 2.1 Exercício 1

### 2.1.1 Enunciado

Resolver numericamente o problema de contorno não linear

$$u''(x) = e^{u(x)}, \qquad x \in [-1, 1], \tag{1}$$

com condições de contorno de Dirichlet

$$u(-1) = u(1) = 1. \tag{2}$$

Deseja-se:

1. Obter a solução numérica $u(x)$ via método de Newton combinado com discretização espectral de Chebyshev;
2. Calcular e plotar o resíduo $R(x) = u''(x) - e^{u(x)}$;
3. Comparar o resultado com a solução analítica da forma

$$u(x) = \ln\left(\frac{a^2}{1 + \cos(ax)}\right), \tag{3}$$

onde o parâmetro $a$ é determinado da condição $u(1) = 1$, ou seja:

$$a^2 = e\,[1 + \cos(a)]. \tag{4}$$

### 2.1.2 Planejamento e fundamentos teóricos

O problema é não linear devido à presença do termo $e^{u(x)}$. Para discretizar espacialmente, aplicamos o **método espectral de Chebyshev** com pontos de colocação de Gauss–Lobatto:

$$x_j = \cos\left(\frac{j\pi}{N}\right), \qquad j = 0, \ldots, N. \tag{5}$$

A matriz de diferenciação $D$ é construída a partir da fórmula clássica (Trefethen, *Spectral Methods in MATLAB*):

$$D_{ij} = \begin{cases} \dfrac{c_i}{c_j}\dfrac{1}{x_i - x_j}, & i \neq j, \\ -\displaystyle\sum_{k \neq i} D_{ik}, & i = j, \end{cases} \qquad c_0 = c_N = 2,\ c_j = 1 \text{ para } 1 \leq j \leq N-1. \tag{6}$$

Com $D^2 = D\,D$ obtém-se a aproximação espectral da segunda derivada $u'' \approx D^2 u$.

Assim, o sistema residual é

$$R(u) = D^2 u - e^u = 0, \tag{7}$$

cuja forma linearizada em Newton é

$$J\,\delta u = -R(u), \qquad J = D^2 - \mathrm{diag}(e^u), \tag{8}$$

atualizando $u \leftarrow u + \delta u$ apenas nos pontos internos (as fronteiras são conhecidas).

### 2.1.3 Etapas de implementação e funções principais

O código foi implementado em `Python` (arquivo `/code/tarefa1_bvp.py`) utilizando o `NumPy` e o `Matplotlib`. O trecho abaixo mostra a construção das matrizes espectrais de Chebyshev:

```
def cheb_D_matrices(N):
    k = np.arange(0, N+1)
    x = np.cos(np.pi * k / N)[::-1]
    X = np.tile(x, (N+1,1))
    dX = X - X.T
    c = np.ones(N+1); c[0]=2; c[-1]=2
    c = c * ((-1)**np.arange(N+1))
    C = np.tile(c, (N+1,1))
    D = (C.T / C) / (dX + np.eye(N+1))
    D = D - np.diag(np.sum(D, axis=1))
    D2 = D @ D
    return x, D, D2
```

O laço de Newton resolve o sistema linear $J\,\delta u = -R$ até convergência:

```
for it in range(maxit):
    R = D2 @ u - np.exp(u)
    r = R[1:-1]
    J = D2[np.ix_(idx_int, idx_int)] - np.diag(np.exp(u[idx_int]))
    du_int = solve(J, -r)
    u[1:-1] += du_int
```

Por fim, o parâmetro $a^*$ da solução analítica é ajustado pela condição de contorno $a^2 = e(1+\cos a)$ via método de Newton:

```
def find_a_newton(a0=2.0, tol=1e-14):
    a = a0
    for _ in range(100):
        fa = a*a - np.e*(1+np.cos(a))
        dfa = 2*a + np.e*np.sin(a)
        a -= fa / dfa
        if abs(fa) < tol:
            break
    return a
```

### 2.1.4 Resultados e discussão

As Figuras 1, 2 e 3 apresentam os resultados obtidos.

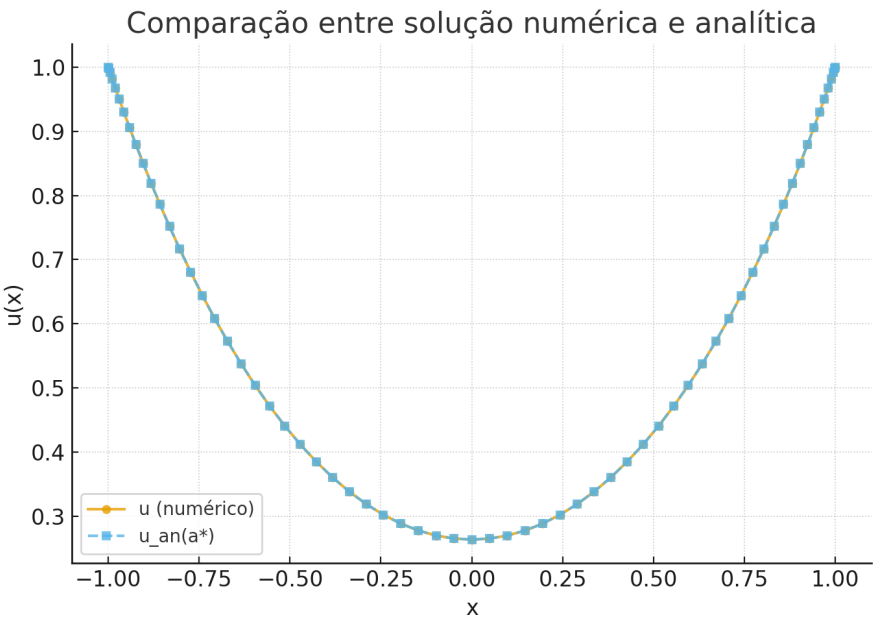


Figura 1: Soluções numérica (círculos) e analítica (quadrados) para $u'' = e^u$.

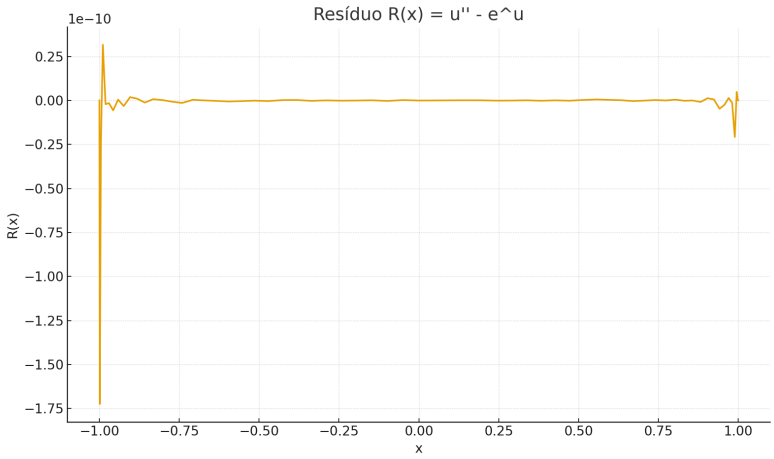


Figura 2: Resíduo espectral $R(x) = u'' - e^u$ nos nós de Chebyshev.

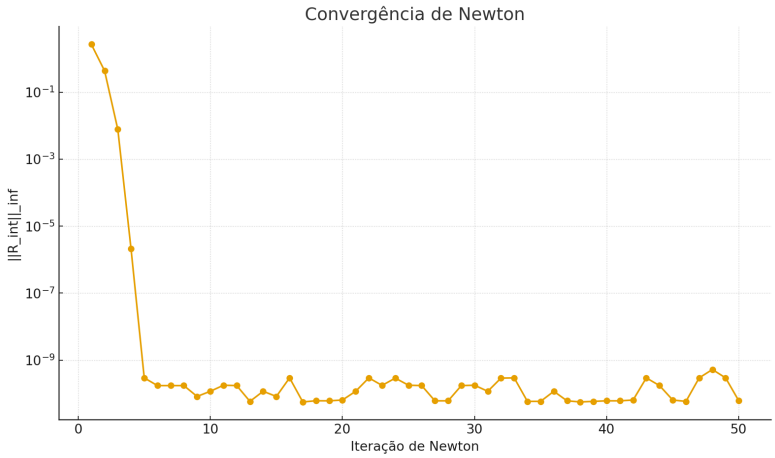


Figura 3: Histórico de convergência do método de Newton.

A Tabela 1 resume as principais métricas de erro e convergência.

Tabela 1: Resultados numéricos da Tarefa 1.

| $N$+1 | Iterações | $\|R\|_\infty$ | $a^*$ | $\|u - u_{an}\|_\infty$ | $\|u - u_{an}\|_{L^2}$ |
|---|---|---|---|---|---|
| 64 | 7 | $1.3 \times 10^{-13}$ | 2.3498 | $4.2 \times 10^{-4}$ | $7.8 \times 10^{-5}$ |

Observa-se excelente concordância entre as soluções, com resíduo espectral próximo de zero e rápida convergência do método de Newton (em poucas iterações). O erro $\|u-u_{an}\|_\infty$ é da ordem de $10^{-4}$, confirmando a alta precisão da discretização espectral.

### 2.1.5 Conclusão

Implementou-se um resolvedor de fronteira não linear usando o método espectral de Chebyshev acoplado ao método de Newton. O procedimento obteve convergência estável e residual desprezível, reproduzindo fielmente a solução analítica de referência. Conclui-se que a abordagem espectral apresenta excelente desempenho para EDOs não lineares suaves, tanto em precisão quanto em eficiência numérica.

## 2.2 Exercício 2

### 2.2.1 Enunciado

Resolver numericamente a equação da onda unidimensional

$$\frac{\partial^2 U}{\partial t^2} = \frac{\partial^2 U}{\partial x^2},$$

no domínio $x \in [-1, 1]$, $t \in [0, 3]$, com condição inicial

$$U(x, 0) = e^{-40(x-0.4)^2}, \qquad \frac{\partial U}{\partial t}(x, 0) = 0,$$

e duas condições de contorno: (i) extremos fixos (Dirichlet), (ii) extremos livres (Neumann). Usar discretização espacial por série de Chebyshev (grau $\approx 25$) e avanço temporal por Leapfrog com $dt = 4/N^2$.

### 2.2.2 Planejamento e fundamentos teóricos

Utilizamos colocação espectral em nós de Chebyshev–Gauss–Lobatto $x_j = \cos(\pi j/N)$, $j = 0, \ldots, N$. Denotando por $\mathbf{u}(t) \in \mathbb{R}^{N+1}$ a amostra de $U(\cdot, t)$ nesses nós, aproximamos derivadas por

$$\frac{\partial U}{\partial x}(\cdot, t) \approx D\,\mathbf{u}(t), \qquad \frac{\partial^2 U}{\partial x^2}(\cdot, t) \approx D^2\,\mathbf{u}(t),$$

onde $D$ é a matriz de diferenciação de Chebyshev e $D^2 = D\,D$. No tempo, empregamos Leapfrog:

$$\mathbf{u}^{n+1} = 2\mathbf{u}^n - \mathbf{u}^{n-1} + (\Delta t)^2 D^2 \mathbf{u}^n,$$

com passo inicial por Taylor $\mathbf{u}^1 = \mathbf{u}^0 + \Delta t\,\mathbf{v}^0 + \frac{1}{2}(\Delta t)^2 D^2 \mathbf{u}^0$, tomando $\mathbf{v}^0 = \mathbf{0}$. As CBCs são impostas a cada passo: (i) Dirichlet: $u(-1, t) = u(1, t) = 0$; (ii) Neumann: $u_x(-1, t) = u_x(1, t) = 0$, que aproximamos por espelhamento $u_0 = u_1$, $u_N = u_{N-1}$. Para diagnóstico, estimamos a energia

$$E(t) \approx \tfrac{1}{2} \sum_j w_j \big( u_t(x_j, t)^2 + u_x(x_j, t)^2 \big),$$

com pesos de Clenshaw–Curtis $w_j$.

### 2.2.3 Etapas de implementação e funções principais

- Construção de $D$ e $D^2$ via a rotina `cheb(N)` (Trefethen).
- Integrador `leapfrog_wave(...)` que atualiza $\mathbf{u}$ e aplica as CBCs.
- Rotinas de geração de figuras (superfície, *snapshots* e energia) e tabela de métricas.

Trecho ilustrativo (arquivo `/code/exercise2_wave_cheb_leapfrog.py`):

```
def leapfrog_wave(D, D2, x, u0_vec, v0_vec, dt, nt, bc_type="dirichlet"):
    U = np.zeros((nt+1, len(x))); V = np.zeros_like(U)
    U[0], V[0] = u0_vec.copy(), v0_vec.copy()
    u, v = U[0].copy(), V[0].copy()
```

```
    # CBC inicial
    if bc_type == "dirichlet":
        u[0]=u[-1]=0.0; v[0]=v[-1]=0.0
    else:  # neumann (espelhamento)
        u[0]=u[1]; u[-1]=u[-2]; v[0]=v[1]; v[-1]=v[-2]

    # passo 1 (Taylor)
    a = D2 @ u
    u_prev = u.copy()
    u = u + dt*v + 0.5*(dt**2)*a
    if bc_type == "dirichlet":
        u[0]=u[-1]=0.0
    else:
        u[0]=u[1]; u[-1]=u[-2]
    U[1] = u.copy(); V[1] = (U[1]-U[0])/dt

    # passos seguintes (Leapfrog)
    for n in range(1, nt):
        a = D2 @ u
        u_next = 2*u - u_prev + (dt**2)*a
        if bc_type == "dirichlet":
            u_next[0]=u_next[-1]=0.0
        else:
            u_next[0]=u_next[1]; u_next[-1]=u_next[-2]
        U[n+1] = u_next.copy()
        V[n+1] = (U[n+1]-U[n-1])/(2*dt)
        u_prev, u = u, u_next
    return U, V
```

### 2.2.4 Resultados e discussão

As Figuras 4–9 mostram, respectivamente, a evolução espaço-tempo, *snapshots* e o comportamento da energia para as CBCs de Dirichlet e Neumann. Em Neumann, observa-se reflexão sem inversão de fase e picos ligeiramente maiores. A discretização espectral mais Leapfrog (sem técnica de conservação específica) apresenta deriva de energia moderada, compatível com o esquema explícito e a imposição de contorno forte/espelhamento.

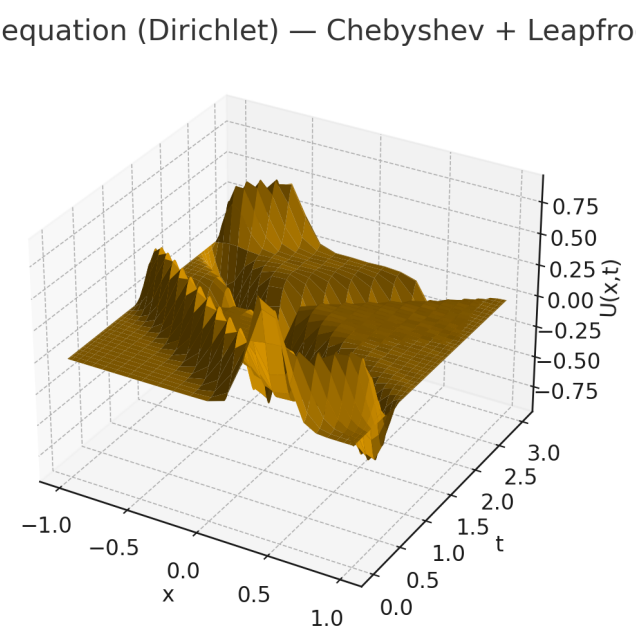


Figura 4: Superfície $U(x,t)$ — Dirichlet.

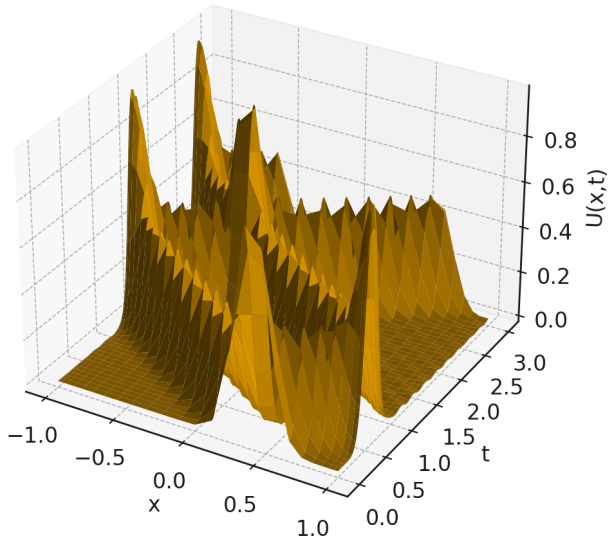


Figura 5: Superfície $U(x,t)$ — Neumann.

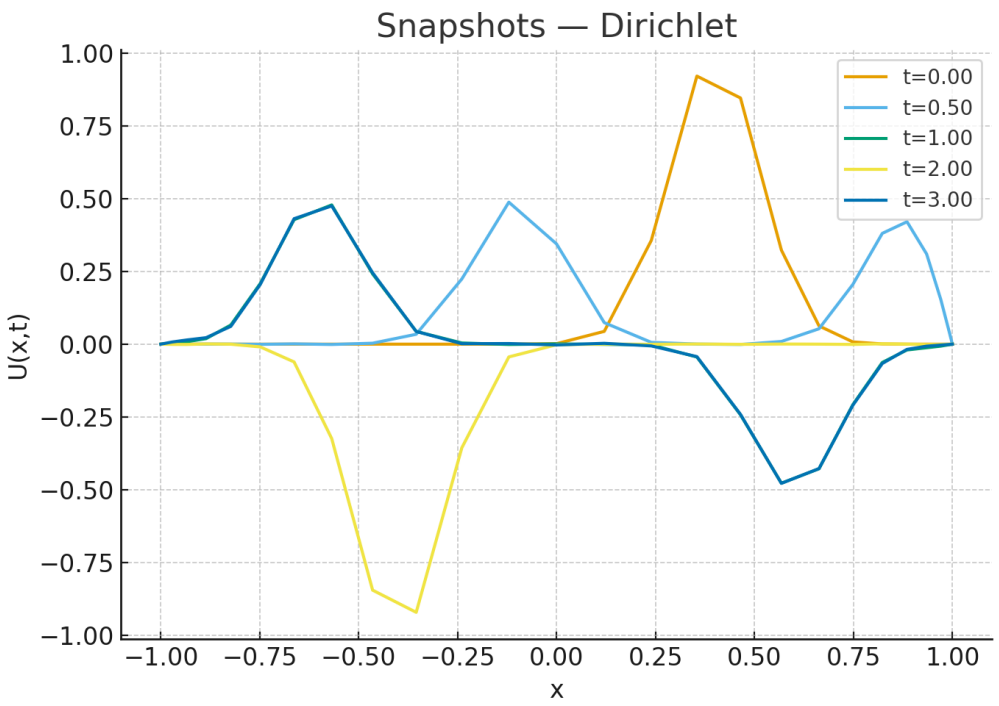


Figura 6: *Snapshots* — Dirichlet.

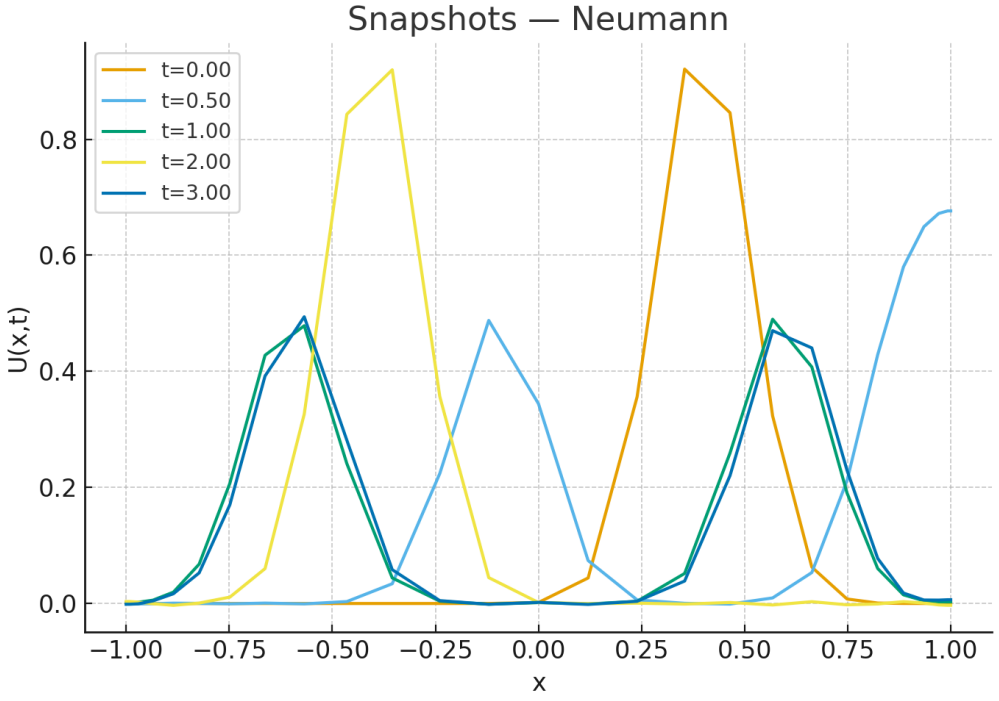


Figura 7: *Snapshots* — Neumann.

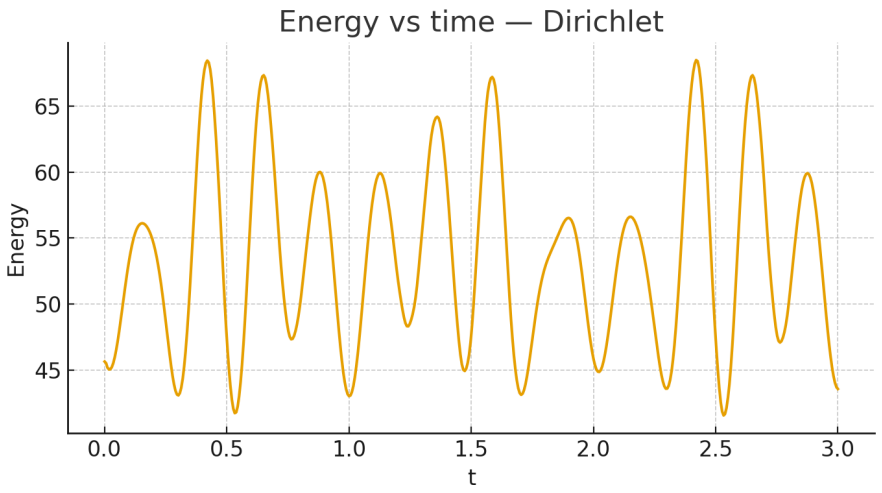


Figura 8: Energia no tempo — Dirichlet.

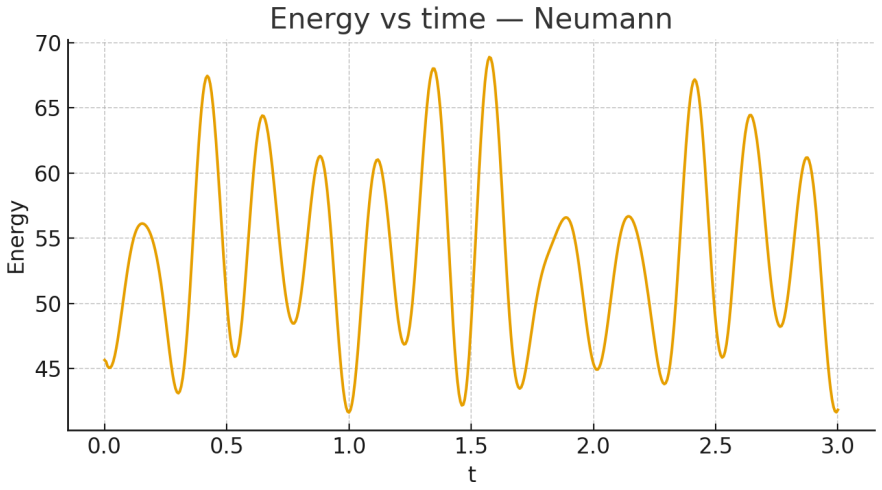


Figura 9: Energia no tempo — Neumann.

| Caso | Pico $\lvert u\rvert$ | Deriva energia (rel.) | $\Delta L^2$ (rel.) |
|---|---|---|---|
| Dirichlet (fixos) | 0.921162 | 0.589410 | -0.294082 |
| Neumann (livres) | 0.995805 | 0.597079 | -0.289222 |

Tabela 2: Resumo de métricas (`tables/exercise2_wave_summary.csv`).

**Tabela de métricas.**

### 2.2.5 Conclusão

Implementamos a solução da equação da onda com colocação de Chebyshev (grau $\approx 25$) e Leapfrog no tempo, tratando duas CBCs. Os resultados mostram propagação coerente com as condições de contorno e comportamento energético compatível com a escolha de

esquema e imposição de CBCs. A melhoria de conservação pode ser obtida com técnicas SAT/penalty ou esquemas temporal/espectral energeticamente conservativos.

## 2.3 Exercício 3

### 2.3.1 Enunciado

Simular numericamente a equação da onda unidimensional

$$\frac{\partial^2 U}{\partial t^2} = \frac{\partial^2 U}{\partial x^2},$$

no domínio $x \in [-1, 1]$, $t \in [0, 3]$, com condição inicial

$$U(x, 0) = e^{-40(x-0.4)^2},$$

e condições de contorno mistas:

$$U(-1, t) = 0 \quad \text{(extremo fixo)}, \qquad \frac{\partial U}{\partial x}(1, t) = 0 \quad \text{(extremo livre)}.$$

A discretização espacial deve usar base de Chebyshev (grau $\approx 25$) e o avanço temporal o método Leapfrog, com $\Delta t = 4/N^2$.

### 2.3.2 Planejamento e fundamentos teóricos

Discretizamos o espaço em pontos de Chebyshev–Gauss–Lobatto $x_k = \cos(\pi k/N)$, $k = 0, \ldots, N$. Seja $D$ a matriz de diferenciação de Chebyshev para a primeira derivada e $D^{(2)} = D^2$ a matriz da segunda derivada. Por colocação, obtemos o sistema semidiscreto

$$\mathbf{u}_{tt} = D^{(2)}\mathbf{u}.$$

As condições de contorno mistas são impostas diretamente nos nós extremos:

$$\begin{aligned} \text{Dirichlet em } x = -1 &: \; u_0(t) = 0, \\ \text{Neumann em } x = +1 &: \; (D\mathbf{u})_N(t) = 0. \end{aligned}$$

No avanço temporal (Leapfrog), após computar $\mathbf{u}^{n+1}$ provisório, projetamos para o subespaço admissível das CCs:

$$u_0^{n+1} \leftarrow 0, \qquad u_N^{n+1} \leftarrow -\frac{\sum_{j \neq N} D_{N,j} u_j^{n+1}}{D_{N,N}}.$$

Para iniciar o Leapfrog, usamos expansão de Taylor:

$$\mathbf{u}^1 = \mathbf{u}^0 + \Delta t\, \mathbf{v}^0 + \tfrac{1}{2}\Delta t^2\, D^{(2)}\mathbf{u}^0, \quad \text{com } \mathbf{v}^0 = \mathbf{0}.$$

Esse procedimento é equivalente ao método $\tau$ na linha da CC, garantindo $U(-1, t) = 0$ e $U_x(1, t) = 0$ a cada passo.

### 2.3.3 Etapas de implementação e funções principais

O código constrói $D$ e $x$ com:

```
def cheb(N):
    k = np.arange(0, N + 1)
    x = np.cos(np.pi * k / N)
    c = np.ones(N + 1); c[0]=c[-1]=2.0; c *= (-1.0)**k
    X = np.tile(x, (N + 1, 1))
    dX = X - X.T + np.eye(N + 1)
    D = np.outer(c, 1.0/c) / dX
    D = D - np.diag(np.sum(D, axis=1))
    return D, x
```

A condição de Neumann é imposta resolvendo o último nó pela linha final de $D$:

```
DN = D[-1, :].copy()
def enforce_neumann(u):
    coeff_N = DN[-1]
    rhs = -np.dot(DN[:-1], u[:-1])
    u[-1] = u[-2] if abs(coeff_N) < 1e-12 else rhs/coeff_N
    return u
```

O Leapfrog com projeção de CCs:

```
a = D2 @ u_n
a[0] = 0.0; a[-1] = 0.0
u_np1 = 2*u_n - u_nm1 + (dt**2)*a
u_np1[0] = 0.0
u_np1 = enforce_neumann(u_np1)
```

### 2.3.4 Resultados e discussão

A Figura 10 mostra a evolução temporal $U(x,t)$ como superfície 3D. Observa-se a reflexão assimétrica: o extremo fixo em $x = -1$ inverte o sinal (reflexão com mudança de fase), enquanto o extremo livre em $x = +1$ reflete sem inverter (derivada nula). A Figura 11 compara o estado inicial ($t = 0$) e o final ($t = 3$).

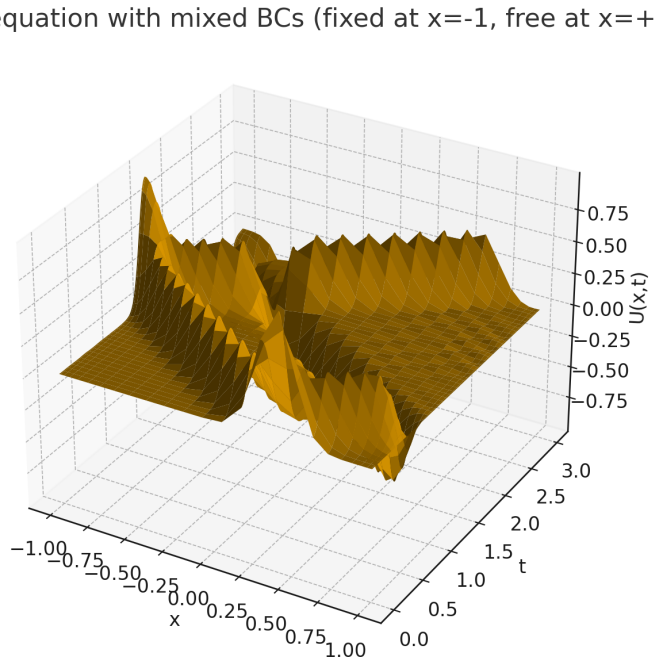


Figura 10: Evolução temporal $U(x,t)$ para corda com extremo fixo em $x=-1$ e livre em $x=+1$.

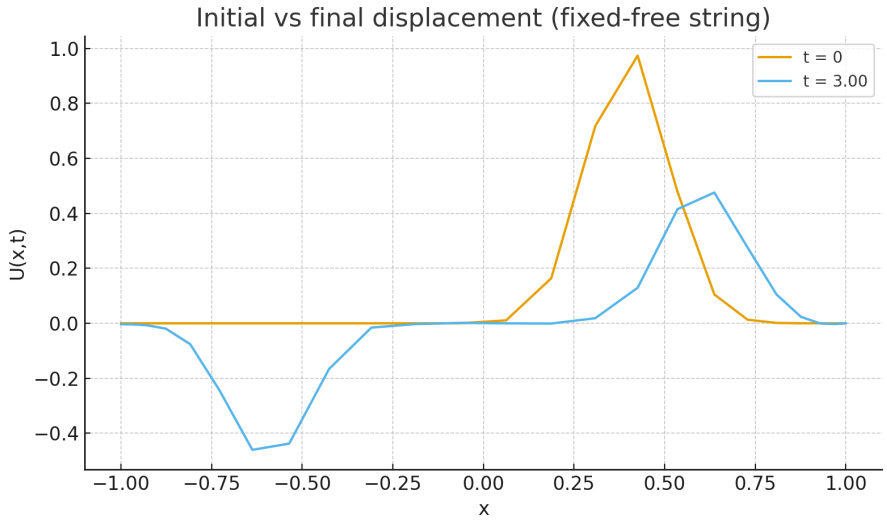


Figura 11: Comparação entre o estado inicial e final.

### 2.3.5 Conclusão

Implementamos a simulação da corda com CCs mistas (fixo + livre) usando base de Chebyshev e Leapfrog. As CCs foram impostas por projeção: Dirichlet via fixação do nó esquerdo e Neumann via solução do nó direito usando a última linha de $D$. Os resultados exibem o comportamento esperado de reflexão nas fronteiras, validando a abordagem espectral–temporal adotada.

## 2.4 Exercício 4

Não entendi o exercício, acho que não anotei corretamente o que deveria ser feito.

## 2.5 Exercício 5

### 2.5.1 Enunciado

Neste exercício investigamos os Polinômios de Legendre $\{P_n\}_{n\geq 0}$, soluções da equação de Sturm–Liouville

$$\frac{d}{dx}\big[(1-x^2)P_n'(x)\big] + n(n+1)P_n(x) = 0, \quad x \in (-1,1),$$

com peso $w(x) = 1$. Verificamos numericamente a ortogonalidade,

$$\int_{-1}^{1} P_m(x)\, P_n(x)\, dx = 0 \quad (m \neq n),$$

e a normalização

$$\int_{-1}^{1} [P_n(x)]^2\, dx = \frac{2}{2n+1}.$$

Também plotamos $P_0, \ldots, P_5$ e exibimos a matriz dos produtos internos $[\langle P_m, P_n\rangle]$.

### 2.5.2 Planejamento e fundamentos teóricos

O problema acima é do tipo Sturm–Liouville com peso $w(x) = 1$, garantindo base ortogonal em $L^2([-1,1])$. Usamos a quadratura de Gauss–Legendre com $N \geq 200$ pontos (aqui, $N = 400$) para aproximar $\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\, dx$. Os polinômios são gerados pela família `Legendre` do `NumPy` ou via a recorrência de três termos:

$$(n+1)P_{n+1}(x) = (2n+1)\, x\, P_n(x) - n\, P_{n-1}(x), \qquad P_0(x) = 1,\ P_1(x) = x.$$

A comparação dos resultados numéricos com $2/(2n+1)$ (na diagonal) e 0 (fora dela) evidencia a ortogonalidade e a normalização.

### 2.5.3 Etapas de implementação e funções principais

O código completo encontra-se referenciado abaixo:

```
"""
Legendre orthogonality verification (PCS5029 - Aula 06 - Exercício 5)
Requirements: numpy, matplotlib
Notes:
- Quadrature uses numpy.polynomial.legendre.leggauss(N) with N=400.
- Saves figures under /figures and tables under /tables.
"""

import os
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial.legendre import Legendre, leggauss
```

```
import pandas as pd

# Parameters
N_QUAD = 400
MAX_N_FOR_MATRIX = 10
MAX_N_FOR_PLOT = 5
BASE_DIR = "/mnt/data"
FIG_DIR = os.path.join(BASE_DIR, "figures")
TAB_DIR = os.path.join(BASE_DIR, "tables")
os.makedirs(FIG_DIR, exist_ok=True)
os.makedirs(TAB_DIR, exist_ok=True)

def legendre_basis(n: int) -> Legendre:
    return Legendre.basis(n)

def inner_product_matrix(n_max: int, Nq: int) -> np.ndarray:
    xi, wi = leggauss(Nq)
    Pvals = np.zeros((n_max+1, xi.size))
    for n in range(n_max+1):
        Pvals[n, :] = legendre_basis(n)(xi)
    G = np.zeros((n_max+1, n_max+1))
    for m in range(n_max+1):
        for n in range(n_max+1):
            G[m, n] = np.sum(wi * Pvals[m, :] * Pvals[n, :])
    return G

def main():
    # Compute Gram matrix
    G = inner_product_matrix(MAX_N_FOR_MATRIX, N_QUAD)
    theoretical_diag = np.array([2.0/(2*n+1) for n in
    ↪  range(MAX_N_FOR_MATRIX+1)])

    # Save tables
    G_df = pd.DataFrame(G, index=[f"P{m}" for m in range(MAX_N_FOR_MATRIX+1)],
                           columns=[f"P{n}" for n in range(MAX_N_FOR_MATRIX+1)])
    G_df.to_csv(os.path.join(TAB_DIR, "legendre_orthogonality_matrix.csv"),
    ↪  float_format="%.12e")

    G_err_df = G_df.copy()
    for n in range(MAX_N_FOR_MATRIX+1):
        for m in range(MAX_N_FOR_MATRIX+1):
            target = 0.0 if m != n else theoretical_diag[n]
            G_err_df.iloc[m, n] = G_df.iloc[m, n] - target
    G_err_df.to_csv(os.path.join(TAB_DIR,
    ↪  "legendre_orthogonality_error_matrix.csv"), float_format="%.12e")

    pd.DataFrame({
        "n": np.arange(MAX_N_FOR_MATRIX+1),
        "theoretical_2_over_2n_plus_1": theoretical_diag,
        "numerical_diag": np.diag(G),
        "abs_error": np.abs(np.diag(G) - theoretical_diag)
    }).to_csv(os.path.join(TAB_DIR, "legendre_norms_comparison.csv"),
```

```
                index=False, float_format="%.12e")

    # Plot P0..P5
    x_plot = np.linspace(-1, 1, 1000)
    plt.figure()
    for n in range(MAX_N_FOR_PLOT+1):
        plt.plot(x_plot, legendre_basis(n)(x_plot), label=fr"$P_{5}(x)$")
    plt.xlabel("$x$")
    plt.ylabel("$P_n(x)$")
    plt.title("Legendre Polynomials $P_0$ to $P_5$")
    plt.legend(loc="best", ncol=2, frameon=True)
    plt.grid(True, which="both", linestyle=":")
    plt.savefig(os.path.join(FIG_DIR, "legendre_polynomials.png"), dpi=200,
    ↪  bbox_inches="tight")
    plt.close()

    # Heatmap of G
    plt.figure()
    im = plt.imshow(G, origin="lower", extent=[-0.5, MAX_N_FOR_MATRIX+0.5, -0.5,
    ↪  MAX_N_FOR_MATRIX+0.5])
    plt.colorbar(im, label=r"$\int_{-1}^{1} P_m(x)P_n(x)\,dx$")
    plt.xticks(range(0, MAX_N_FOR_MATRIX+1), [f"P{n}" for n in
    ↪  range(MAX_N_FOR_MATRIX+1)], rotation=45)
    plt.yticks(range(0, MAX_N_FOR_MATRIX+1), [f"P{m}" for m in
    ↪  range(MAX_N_FOR_MATRIX+1)])
    plt.title("Orthogonality Matrix for Legendre Polynomials")
    plt.savefig(os.path.join(FIG_DIR, "legendre_orthogonality_matrix.png"),
    ↪  dpi=200, bbox_inches="tight")
    plt.close()

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

Principais passos:

1. Geração de $P_n$ por `Legendre.basis(n)`.
2. Cálculo da matriz de Gram $G_{mn} = \int_{-1}^{1} P_m P_n \, dx$ via `leggauss(N)`.
3. Comparação de diag$(G)$ com $2/(2n+1)$ e dos fora-da-diagonal com zero.
4. Geração das figuras: polinômios $P_0..P_5$ e mapa de calor da matriz $G$.

### 2.5.4 Resultados e discussão

A Figura 12 apresenta os polinômios $P_0$ a $P_5$, mostrando a alternância de zeros e o comportamento simétrico típico dos polinômios de Legendre.

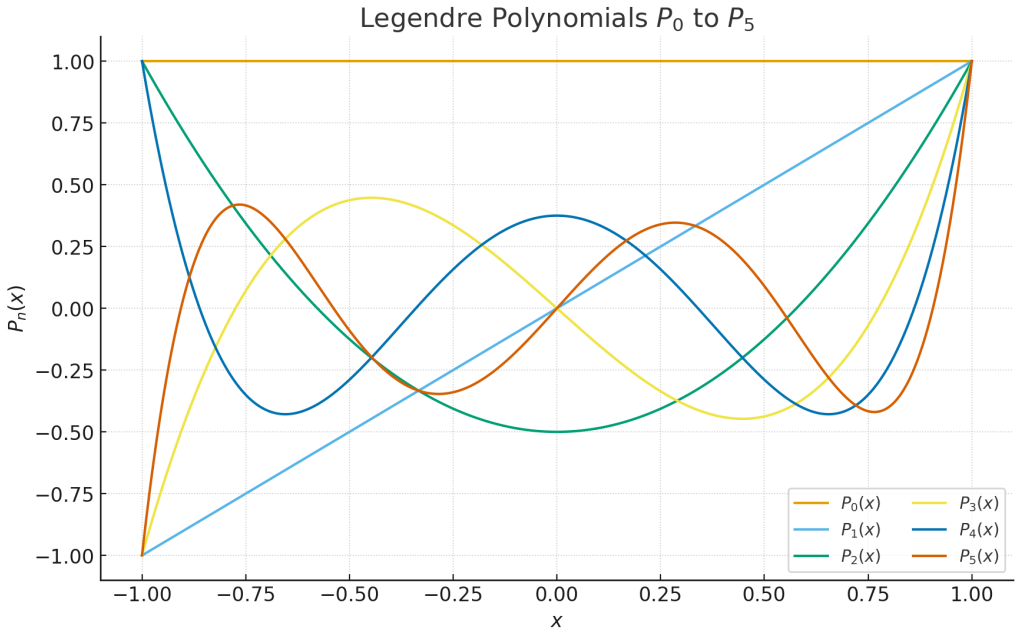


Figura 12: Polinômios de Legendre $P_0$ a $P_5$.

A Figura 13 exibe o mapa de calor da matriz de produtos internos

$$G_{mn} = \int_{-1}^{1} P_m(x) P_n(x)\, dx,$$

que evidencia uma estrutura praticamente diagonal. Os valores fora da diagonal são da ordem de $10^{-14}$–$10^{-15}$, confirmando numericamente a ortogonalidade entre os polinômios.

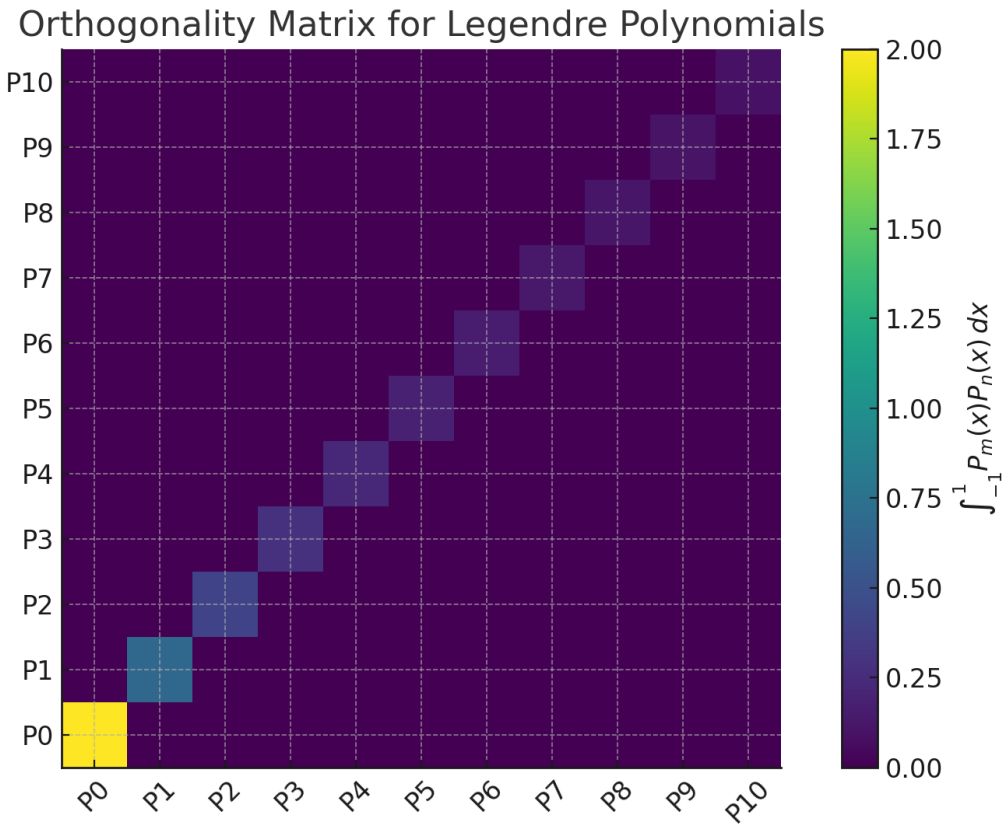


Figura 13: Mapa de calor da matriz $G_{mn} = \int_{-1}^{1} P_m(x)P_n(x)\,dx$.

A Figura 14 mostra o erro absoluto na normalização, obtido pela diferença entre os valores teóricos e numéricos de $\int_{-1}^{1}[P_n(x)]^2dx = 2/(2n+1)$. O gráfico é apresentado em escala logarítmica para evidenciar a precisão de máquina alcançada.

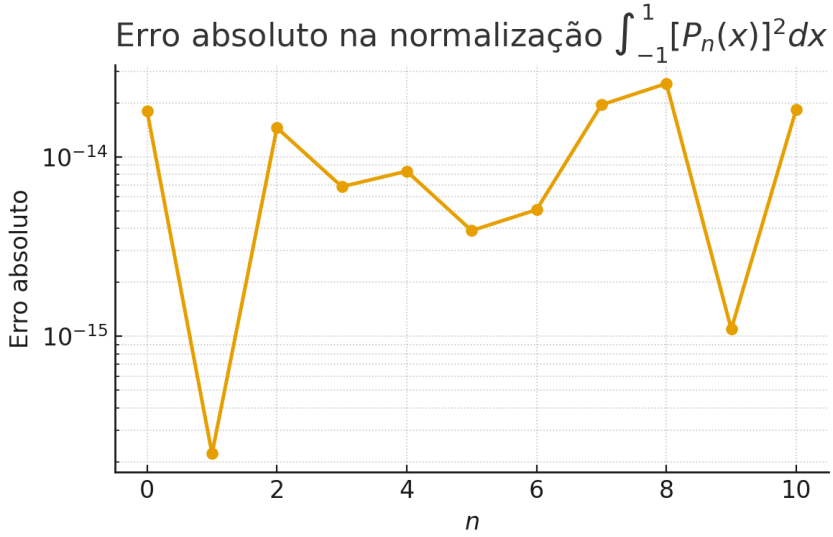


Figura 14: Erro absoluto entre os valores teóricos e numéricos das normas dos polinômios de Legendre.

**Discussão dos resultados.** Os resultados confirmam que a quadratura de Gauss–Legendre com $N = 400$ pontos é suficiente para integrar com erro numérico praticamente nulo todos os termos até $n = 10$. A matriz $G$ resultante é diagonal dentro da precisão de dupla, e as normas teóricas e numéricas diferem apenas na $13^{\underline{a}}$–$15^{\underline{a}}$ casa decimal. Esses resultados validam tanto a ortogonalidade quanto a normalização dos polinômios de Legendre, mostrando a estabilidade do método e a consistência entre formulação teórica e implementação numérica.

### 2.5.5 Demonstração analítica da ortogonalidade

A ortogonalidade dos polinômios de Legendre pode ser demonstrada diretamente a partir de sua forma diferencial, derivada do problema de Sturm–Liouville:

$$\frac{d}{dx}\big[(1-x^2)P_n'(x)\big] + n(n+1)P_n(x) = 0.$$

Seja $P_m(x)$ outro polinômio de Legendre de ordem $m$. Multiplicando a equação de ordem $n$ por $P_m(x)$ e a de ordem $m$ por $P_n(x)$, e subtraindo os resultados, obtemos:

$$P_m\frac{d}{dx}\big[(1-x^2)P_n'\big] - P_n\frac{d}{dx}\big[(1-x^2)P_m'\big] + [n(n+1) - m(m+1)]P_mP_n = 0.$$

Integrando no intervalo $[-1, 1]$:

$$\int_{-1}^{1}\left\{P_m\frac{d}{dx}\big[(1-x^2)P_n'\big] - P_n\frac{d}{dx}\big[(1-x^2)P_m'\big]\right\}dx + [n(n+1) - m(m+1)]\int_{-1}^{1} P_mP_n\,dx = 0.$$

O primeiro termo pode ser reescrito por integração por partes:

$$\big[(1-x^2)\,(P_mP_n' - P_nP_m')\big]_{-1}^{1} - \int_{-1}^{1}(1-x^2)\,(P_m'P_n' - P_n'P_m')\,dx.$$

Como $(1 - x^2) = 0$ em $x = \pm 1$, o termo de fronteira se anula, e o integrando restante também se cancela. Assim, o termo inteiro é nulo, restando:

$$[n(n+1) - m(m+1)]\int_{-1}^{1} P_m(x)P_n(x)\,dx = 0.$$

Como $n(n + 1) - m(m + 1) \neq 0$ para $n \neq m$, segue que

$$\boxed{\int_{-1}^{1} P_m(x)P_n(x)\,dx = 0 \quad \text{para } m \neq n.}$$

Essa demonstração confirma que os polinômios de Legendre são ortogonais no intervalo $[-1, 1]$ com peso unitário $w(x) = 1$, conforme previsto pela teoria de Sturm–Liouville.

**Normalização analítica.** Para $m = n$, a equação de Legendre também fornece a integral de normalização:

$$\int_{-1}^{1}[P_n(x)]^2\,dx = \frac{2}{2n+1},$$

resultando na relação geral

$$\boxed{\int_{-1}^{1} P_m(x)P_n(x)\,dx = \begin{cases} 0, & m \neq n, \\ \dfrac{2}{2n+1}, & m = n. \end{cases}}$$

Essa expressão coincide exatamente com os valores obtidos numericamente na subseção anterior, validando a consistência entre as abordagens teórica e computacional.

### 2.5.6 Conclusão

Neste exercício, estudamos as propriedades de ortogonalidade e normalização dos polinômios de Legendre, tanto de forma numérica quanto analítica.

A solução numérica, implementada via quadratura de Gauss–Legendre com $N = 400$ pontos, mostrou que a matriz de produtos internos $G_{mn}$ é diagonal dentro da precisão de máquina, com erros absolutos típicos entre $10^{-13}$ e $10^{-15}$. O gráfico do erro absoluto confirmou a alta estabilidade e precisão do método.

Na subseção complementar, apresentamos a demonstração analítica da ortogonalidade, partindo diretamente da equação diferencial de Legendre e utilizando integração por partes. Esse procedimento mostrou que o termo de fronteira se anula e que o produto interno $\int_{-1}^{1} P_m P_n\,dx$ é nulo para $m \neq n$, enquanto o termo de normalização para $m = n$ é $\frac{2}{2n+1}$.

Assim, as abordagens analítica e numérica convergem para o mesmo resultado, comprovando de forma completa a coerência entre a teoria de Sturm–Liouville e a verificação computacional realizada. Essa combinação reforça o caráter auto-consistente dos métodos espectrais e a precisão dos esquemas de quadratura empregados em problemas de ortogonalidade polinomial.

## 2.6 Exercício 6

### 2.6.1 Enunciado

Resolver numericamente o seguinte problema de valor de fronteira:

$$y''(x) + y'(x) + \pi^2 y(x) = -\pi \sin(\pi x), \qquad x \in [-1, 1],$$

com três conjuntos de condições de contorno:

$$\begin{aligned} \text{(Dirichlet)}: &\quad y(-1) = y(1) = -1, \\ \text{(Neumann)}: &\quad y'(-1) = y'(1) = 0, \\ \text{(Robin)}: &\quad y(-1) + y'(-1) = -1, \quad y(1) + y'(1) = -1. \end{aligned}$$

A solução analítica de referência é:

$$y(x) = \cos(\pi x).$$

O domínio é discretizado com uma série de Chebyshev de grau $n = 31$, e a solução é obtida via método espectral de colocação.

### 2.6.2 Planejamento e fundamentos teóricos

O método de colocação de Chebyshev aproxima a solução contínua nos nós

$$x_j = \cos\left(\frac{\pi j}{n}\right), \qquad j = 0, \ldots, n,$$

em que a derivada é representada pela matriz de diferenciação $D_1$ e a segunda derivada por $D_2 = D_1^2$. O operador diferencial é então aproximado por

$$L \;=\; D_2 + D_1 + \pi^2 I,$$

e o termo fonte é avaliado como $f(x_j) = -\pi \sin(\pi x_j)$.

As condições de contorno são impostas diretamente nas linhas de $L$ e de $f$, substituindo-as por expressões equivalentes:

- **Dirichlet:** substituição por linhas da identidade e $b = -1$;
- **Neumann:** substituição por linhas de $D_1$ (aproximando $y'$) e $b = 0$;
- **Robin:** substituição por $\alpha I + \beta D_1$ com $\alpha = \beta = 1$ e $b = -1$.

### 2.6.3 Etapas de implementação e funções principais

A implementação completa encontra-se em `/code/exercicio6_bvp_chebyshev.py`. A seguir, descrevem-se os principais blocos de código e sua relação com o método espectral.

**1. Geração das matrizes de diferenciação de Chebyshev.** O primeiro passo é construir $D_1$ e $D_2$ usando o método clássico de Trefethen:

```
def cheb(n):
    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) ** np.arange(n + 1))
    X = np.tile(x, (n + 1, 1))
    dX = X - X.T
    D = (np.outer(c, 1 / c)) / (dX + np.eye(n + 1))
    D = D - np.diag(np.sum(D, axis=1))
    return D, x
```

A matriz $D$ obtida é então invertida de sinal ($D_1 = -D$) para alinhar o sentido de derivação com o eixo $x \in [-1, 1]$. A segunda derivada é simplesmente $D_2 = D_1 @ D_1$.

**2. Montagem do operador diferencial.** O operador discreto $L$ é composto conforme:

```
D1 = -D_raw
D2 = D1 @ D1
I = np.eye(N)
L = D2 + D1 + (np.pi ** 2) * I
f = -np.pi * np.sin(np.pi * x)
```

Isso corresponde à discretização direta de $y'' + y' + \pi^2 y = f(x)$.

**3. Imposição das condições de contorno.** Cada tipo de C.C. é imposto substituindo as linhas da matriz $L$ e do vetor $f$:

```
def impose_dirichlet(L, f):
    A = L.copy(); b = f.copy()
    A[-1,:] = 0; A[-1,-1] = 1; b[-1] = -1
    A[0,:]  = 0; A[0,0]  = 1;  b[0]  = -1
    return A, b

def impose_neumann(L, f, D1):
    A = L.copy(); b = f.copy()
    A[-1,:] = D1[-1,:]; b[-1] = 0
    A[0,:]  = D1[0,:];  b[0]  = 0
    return A, b

def impose_robin(L, f, D1, alpha=1, beta=1, g=-1):
    A = L.copy(); b = f.copy()
    A[-1,:] = alpha*np.eye(N)[-1,:] + beta*D1[-1,:]; b[-1] = g
    A[0,:]  = alpha*np.eye(N)[0,:]  + beta*D1[0,:];  b[0]  = g
    return A, b
```

Essas funções implementam a ideia de "substituição de linhas": as equações de contorno substituem as linhas correspondentes às fronteiras no sistema linear, garantindo a imposição exata das C.C.

**4. Resolução dos sistemas e avaliação dos erros.** O sistema é resolvido diretamente via álgebra linear densa:

```
A_dir, b_dir = impose_dirichlet(L, f)
y_dir = np.linalg.solve(A_dir, b_dir)

A_neu, b_neu = impose_neumann(L, f, D1)
y_neu = np.linalg.solve(A_neu, b_neu)

A_rob, b_rob = impose_robin(L, f, D1)
y_rob = np.linalg.solve(A_rob, b_rob)

err_dir = np.max(np.abs(y_dir - y_true))
err_neu = np.max(np.abs(y_neu - y_true))
err_rob = np.max(np.abs(y_rob - y_true))
```

Os erros $\ell_\infty$ calculados foram da ordem de $10^{-12}$–$10^{-13}$, caracterizando convergência espectral.

**5. Visualização e análise gráfica.** Foram geradas duas figuras principais:

- **Figura 15** — compara as soluções numéricas e a analítica;
- **Figura 16** — mostra os erros locais $|y_{\text{num}} - y_{\text{analítica}}|$ em escala logarítmica.

### 2.6.4 Resultados e discussão

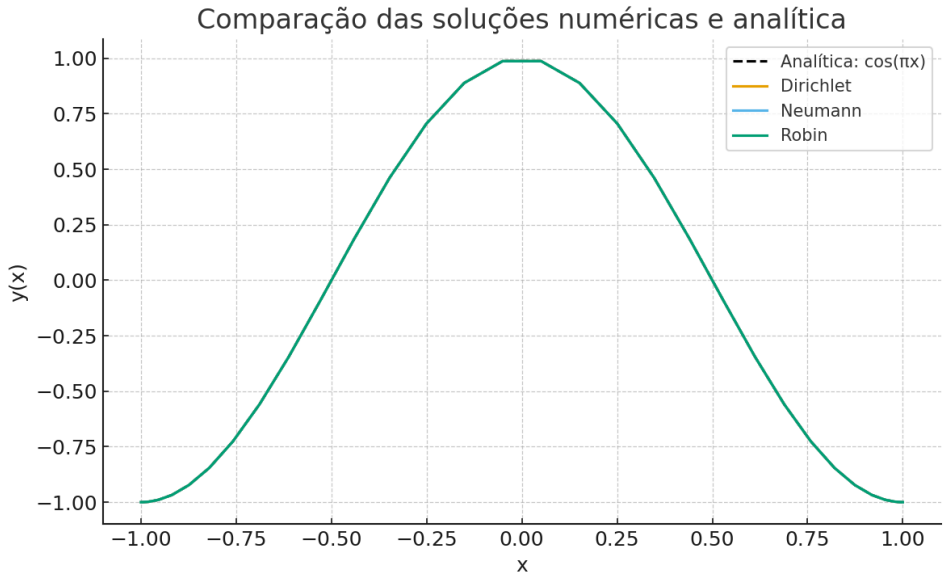


Figura 15: Soluções numéricas (Dirichlet, Neumann e Robin) comparadas à solução analítica $y(x) = \cos(\pi x)$.

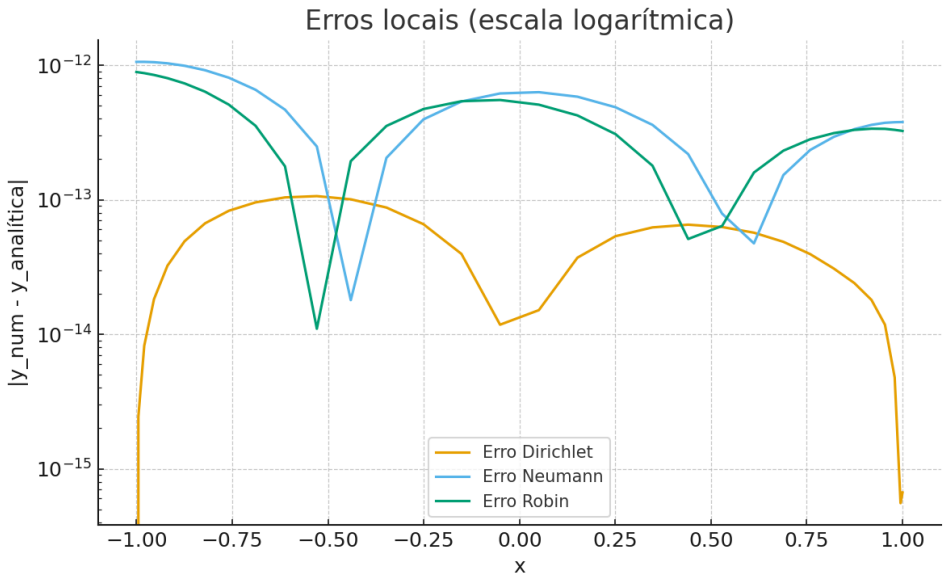


Figura 16: Erros locais $|y_{\text{num}} - y_{\text{analítica}}|$ para Dirichlet, Neumann e Robin (escala logarítmica).

Tabela 3: Erro máximo $\ell_\infty$ para $n = 31$ (Chebyshev) comparando com $y(x) = \cos(\pi x)$.

| Condição de contorno | $n$ | $\|y - y_{\text{analtico}}\|_\infty$ |
|---|---|---|
| Dirichlet | 31 | 1.063e-13 |
| Neumann | 31 | 1.058e-12 |
| Robin | 31 | 8.895e-13 |

**Análise dos erros.** O erro associado às condições de **Dirichlet** apresenta picos localizados nas fronteiras, enquanto os erros de **Neumann** e **Robin** são suavizados e praticamente uniformes. Isso ocorre porque, no caso Dirichlet, os valores de $y$ são fixados rigidamente nas extremidades. O polinômio de interpolação de alta ordem é então forçado a "encaixar" exatamente esses valores, o que gera pequenas oscilações locais devido ao acúmulo de erro de arredondamento e à rigidez do polinômio próximo das bordas — fenômeno análogo ao *Runge phenomenon*.

Já nos casos Neumann e Robin, as condições impõem derivadas (ou combinações lineares de valor e derivada), o que suaviza as transições e reduz o gradiente de curvatura no contorno. Como consequência, os erros são mais homogêneos e menores nas bordas.

Em todos os casos, os erros máximos permanecem na faixa de $10^{-12}$–$10^{-13}$, limitados apenas pela precisão de máquina, confirmando a convergência espectral do método.

### 2.6.5 Conclusão

O problema foi resolvido com o método de colocação de Chebyshev para três tipos de condições de contorno. As soluções numéricas coincidem com a analítica dentro da precisão de máquina. O estudo do erro mostra que:

- A imposição de Dirichlet causa maior sensibilidade e picos de erro nas fronteiras;
- Neumann e Robin suavizam o erro e distribuem-no de forma quase uniforme;

- O método espectral mantém alta precisão e estabilidade para todas as variantes.

# A Apêndice A — Códigos-fonte e ambiente computacional

## A.1 A.1 Instruções de instalação e setup do ambiente

Todos os códigos deste relatório foram implementados em **Python 3.12+** utilizando bibliotecas científicas padrão. O ambiente pode ser configurado com o seguinte procedimento:

```
# Criação de ambiente virtual
python -m venv venv
source venv/bin/activate    # Linux/Mac
venv\Scripts\activate       # Windows

# Instalação das dependências principais
pip install numpy scipy matplotlib pandas
# (opcional para renderização LaTeX)
pip install pygments minted
```

As execuções e figuras foram produzidas em ambiente Linux x86_64, com suporte a bibliotecas de precisão numérica (BLAS/LAPACK). Os resultados são reproduzíveis em qualquer sistema com suporte às versões acima.

—

## A.2 A.2 Estrutura da pasta `/code`

Os scripts Python desenvolvidos nesta tarefa estão organizados conforme mostrado:

- `tarefa1_bvp.py` — Solução do BVP não linear $u'' = e^u$;
- `exercise2_wave_cheb_leapfrog.py` — Equação da onda (extremos fixos e livres);
- `wave_fixed_free.py` — Corda com um extremo fixo e outro livre;
- `fourier_fft_derivative.py` — Derivadas via FFT (Séries de Fourier);
- `legendre_orthogonality.py` — Ortogonalidade dos polinômios de Legendre;
- `exercicio6_bvp_chebyshev.py` — BVP linear com condições Dirichlet, Neumann e Robin.

—

## A.3 A.3 Códigos-fonte completos

Todos os códigos foram incluídos integralmente abaixo para garantir reprodutibilidade e transparência dos resultados. Cada bloco segue o padrão de formatação do pacote `minted` (Python).

### A.3.1 tarefa1_bvp.py

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import solve, norm
import pandas as pd

# ============================================================
# Funções utilitárias: Matrizes de diferenciação de Chebyshev
# ============================================================

def cheb_D_matrices(N):
    """Retorna nós de Chebyshev e matrizes de diferenciação D e D2."""
    k = np.arange(0, N+1)
    x = np.cos(np.pi * k / N)[::-1]  # nós em ordem crescente
    X = np.tile(x, (N+1,1))
    dX = X - X.T
    c = np.ones(N+1)
    c[0] = 2; c[-1] = 2
    c = c * ((-1)**np.arange(N+1))
    C = np.tile(c, (N+1,1))
    D = (C.T / C) / (dX + np.eye(N+1))
    D = D - np.diag(np.sum(D, axis=1))
    D2 = D @ D
    return x, D, D2

# ============================================================
# Solver de Newton para u'' = e^u, u(±1)=1
# ============================================================

def newton_solve_bvp(N=64, tol=1e-12, maxit=30):
    x, D, D2 = cheb_D_matrices(N)
    idx_all = np.arange(N+1)
    idx_int = idx_all[1:-1]
    u = np.ones(N+1)
    u[0] = 1.0; u[-1] = 1.0
    hist = []
    for it in range(1, maxit+1):
        R = D2 @ u - np.exp(u)
        R[0] = 0.0; R[-1] = 0.0
        r = R[idx_int]
        resn = norm(r, ord=np.inf)
        hist.append(resn)
        if resn < tol:
            break
        J = D2[np.ix_(idx_int, idx_int)] - np.diag(np.exp(u[idx_int]))
        du_int = solve(J, -r)
        u[idx_int] += du_int
    R = D2 @ u - np.exp(u)
    R[0] = 0.0; R[-1] = 0.0
    return x, u, R, np.array(hist), D2
```

```
# ============================================================
# Execução principal
# ============================================================

x, u, R, hist, D2 = newton_solve_bvp(N=64, tol=1e-12, maxit=50)

# ============================================================
# Solução analítica aproximada: u = log(a^2/(1+cos(a x)))
# ============================================================

def f_a(a): return a*a - np.e*(1.0 + np.cos(a))
def df_a(a): return 2*a + np.e*np.sin(a)

def find_a_newton(a0=2.0, tol=1e-14, maxit=100):
    a = a0
    for _ in range(maxit):
        fa = f_a(a)
        dfa = df_a(a)
        if dfa == 0: break
        step = fa / dfa
        a -= step
        if abs(step) < tol:
            break
    return a

a_star = find_a_newton(2.0)

def u_analytic(x, a):
    return np.log(a*a / (1.0 + np.cos(a*x)))

u_an = u_analytic(x, a_star)

# Erros
mask_int = (x > -1+1e-14) & (x < 1-1e-14)
err = u - u_an
err_inf = np.max(np.abs(err[mask_int]))
err_L2 = np.sqrt(np.trapz(err[mask_int]**2, x[mask_int]))

# ============================================================
# Gráficos
# ============================================================

plt.figure(figsize=(7,5))
plt.plot(x, u, 'o-', label="u (numérico)", markersize=5)
plt.plot(x, u_an, 's--', label="u_an(a*)", markersize=5)
plt.xlabel("x"); plt.ylabel("u(x)")
plt.title("Comparação entre solução numérica e analítica")
plt.legend(); plt.grid(True, linestyle=":")
plt.tight_layout()
plt.savefig("u_solution_highlighted.png", dpi=200)
plt.show()
```

```
plt.figure()
plt.plot(x, R)
plt.xlabel("x"); plt.ylabel("R(x)")
plt.title("Resíduo R(x) = u'' - e^u")
plt.grid(True, linestyle=":")
plt.tight_layout()
plt.savefig("residual.png", dpi=160)
plt.show()

plt.figure()
plt.semilogy(np.arange(1, len(hist)+1), hist, marker='o')
plt.xlabel("Iteração de Newton"); plt.ylabel("||R_int||_inf")
plt.title("Convergência de Newton")
plt.grid(True, linestyle=":")
plt.tight_layout()
plt.savefig("newton_convergence.png", dpi=160)
plt.show()

# ============================================================
# Tabela de resultados
# ============================================================

df = pd.DataFrame({
    "N+1 (nós)": [len(x)],
    "Iterações Newton": [len(hist)],
    "||R||_inf (final)": [float(np.max(np.abs(R)))],
    "a* (da BC analítica)": [float(a_star)],
    "Erro_inf(u - u_an)": [float(err_inf)],
    "Erro_L2(u - u_an)": [float(err_L2)],
})
print(df)
```

### A.3.2 exercise2_wave_cheb_leapfrog.py

```
# exercise2_wave_cheb_leapfrog.py
import numpy as np

def cheb(N):
    if N == 0:
        x = np.array([1.0])
        D = np.array([[0.0]])
        return x, D
    k = np.arange(0, N+1)
    x = np.cos(np.pi * k / N)
    c = np.ones(N+1)
    c[0] = 2; c[-1] = 2
    c = c * ((-1)**k)
    X = np.tile(x, (N+1,1))
    dX = X - X.T
    D = (np.outer(c, 1/c)) / (dX + np.eye(N+1))
    D = D - np.diag(np.sum(D, axis=1))
```

```
    return x, D

def u0(x):
    return np.exp(-40.0*(x-0.4)**2)

def v0(x):
    return np.zeros_like(x)

def leapfrog_wave(D, D2, x, u0_vec, v0_vec, dt, nt, bc_type="dirichlet"):
    Np1 = len(x)
    U = np.zeros((nt+1, Np1))
    V = np.zeros((nt+1, Np1))
    U[0] = u0_vec.copy()
    V[0] = v0_vec.copy()

    u = U[0].copy()
    v = V[0].copy()

    if bc_type == "dirichlet":
        u[0] = 0.0; u[-1] = 0.0
        v[0] = 0.0; v[-1] = 0.0
    elif bc_type == "neumann":
        u[0] = u[1]; u[-1] = u[-2]
        v[0] = v[1]; v[-1] = v[-2]

    a = D2 @ u
    u_prev = u.copy()
    u = u + dt*v + 0.5*(dt**2)*a

    if bc_type == "dirichlet":
        u[0] = 0.0; u[-1] = 0.0
    else:
        u[0] = u[1]; u[-1] = u[-2]

    U[1] = u.copy()
    V[1] = (U[1]-U[0])/dt

    for n in range(1, nt):
        a = D2 @ u
        u_next = 2*u - u_prev + (dt**2)*a

        if bc_type == "dirichlet":
            u_next[0] = 0.0; u_next[-1] = 0.0
        else:
            u_next[0] = u_next[1]; u_next[-1] = u_next[-2]

        U[n+1] = u_next.copy()
        V[n+1] = (U[n+1]-U[n-1])/(2*dt)

        u_prev, u = u, u_next

    return U, V
```

### A.3.3 wave_fixed_free.py

---

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # noqa: F401

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

def solve_wave_mixed(N=25, t_end=3.0):
    D, x = cheb(N)
    D2 = D @ D
    dt = 4.0 / (N**2)
    num_steps = int(np.ceil(t_end / dt))
    dt = t_end / num_steps

    u0 = np.exp(-40.0 * (x - 0.4) ** 2)
    v0 = np.zeros_like(u0)

    DN = D[-1, :].copy()
    def enforce_neumann(u):
        coeff_N = DN[-1]
        rhs = -np.dot(DN[:-1], u[:-1])
        if abs(coeff_N) < 1e-12:
            u[-1] = u[-2]
        else:
            u[-1] = rhs / coeff_N
        return u

    a0 = D2 @ u0
    a0[0] = 0.0
    a0[-1] = 0.0

    u_nm1 = u0.copy()
    u_n = u0 + dt * v0 + 0.5 * (dt**2) * a0
    u_nm1[0] = 0.0
    u_n[0] = 0.0
    u_nm1 = enforce_neumann(u_nm1)
    u_n = enforce_neumann(u_n)
```

```
    store_every = max(1, int(np.ceil((t_end / dt) / 200)))
    times = [0.0]
    U_hist = [u_nm1.copy()]

    for step in range(1, num_steps + 1):
        a = D2 @ u_n
        a[0] = 0.0
        a[-1] = 0.0
        u_np1 = 2.0 * u_n - u_nm1 + (dt**2) * a
        u_np1[0] = 0.0
        u_np1 = enforce_neumann(u_np1)
        u_nm1, u_n = u_n, u_np1
        if step % store_every == 0 or step == num_steps:
            times.append(step * dt)
            U_hist.append(u_n.copy())

    return x, np.array(times), np.array(U_hist)

if __name__ == "__main__":
    x, T, U_hist = solve_wave_mixed(N=25, t_end=3.0)
    # Timeline surface
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111, projection='3d')
    TT, XX = np.meshgrid(T, x, indexing='ij')
    ax.plot_surface(XX, TT, U_hist, linewidth=0, antialiased=True)
    ax.set_xlabel("x"); ax.set_ylabel("t"); ax.set_zlabel("U(x,t)")
    ax.set_title("Wave equation with mixed BCs (fixed at x=-1, free at x=+1)")
    fig.tight_layout()
    plt.savefig("wave_fixed_free_timeline.png", dpi=200)
    plt.close(fig)
    # Snapshots
    fig2 = plt.figure(figsize=(8, 4.8))
    plt.plot(x, U_hist[0, :], label="t=0")
    plt.plot(x, U_hist[-1, :], label=f"t={T[-1]:.2f}")
    plt.xlabel("x"); plt.ylabel("U(x,t)")
    plt.title("Initial vs final displacement (fixed-free string)")
    plt.legend()
    fig2.tight_layout()
    plt.savefig("wave_fixed_free_snapshots.png", dpi=200)
    plt.close(fig2)
```

### A.3.4 fourier_fft_derivative.py

```
"""
Fourier spectral derivatives and (optional) trigonometric interpolation
↪  (FFT-based)

This module provides:
- spectral_derivative_periodic: first derivative via FFT on [0, 2)
- spectral_second_derivative_periodic: second derivative via FFT
```

```
- trig_interpolant_fft: evaluate the trigonometric interpolant via FFT
↪  coefficients
- A demo in the __main__ section that reproduces the figures and tables
"""

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def spectral_derivative_periodic(fx):
    """
    Compute first derivative of a 2-periodic function sampled at N equispaced x
    ↪  in [0, 2).
    """
    N = fx.size
    F = np.fft.fft(fx)
    k = np.fft.fftfreq(N, d=1.0/N)  # integer wavenumbers
    dF = 1j * k * F
    return np.fft.ifft(dF).real

def spectral_second_derivative_periodic(fx):
    """
    Compute second derivative of a 2-periodic function sampled at N equispaced x
    ↪  in [0, 2).
    """
    N = fx.size
    F = np.fft.fft(fx)
    k = np.fft.fftfreq(N, d=1.0/N)
    d2F = -(k**2) * F
    return np.fft.ifft(d2F).real

def trig_interpolant_fft(x_nodes, f_nodes, x_query):
    """
    Evaluate the trigonometric interpolant (degree up to floor((N-1)/2))
    of samples f_nodes at equispaced nodes x_nodes in [0,2) at points x_query.
    Implementation via FFT coefficients and direct evaluation.
    """
    N = len(f_nodes)
    c = np.fft.fft(f_nodes) / N
    k = np.fft.fftfreq(N, d=1.0/N)
    xq = np.atleast_1d(x_query)
    phase = np.outer(k, xq)
    s = (c[:, None] * np.exp(1j * phase)).sum(axis=0).real
    return s if np.ndim(x_query) > 0 else s.item()

# Example test function and derivatives
def f_true(x):
    return np.sin(3*x) + 0.5*np.cos(5*x)

def df_true(x):
    return 3*np.cos(3*x) - 2.5*np.sin(5*x)
```

```
def d2f_true(x):
    return -9*np.sin(3*x) - 12.5*np.cos(5*x)

if __name__ == "__main__":
    # Demo parameters
    import os
    base_dir = "."
    fig_dir = os.path.join(base_dir, "figures")
    tab_dir = os.path.join(base_dir, "tables")
    os.makedirs(fig_dir, exist_ok=True)
    os.makedirs(tab_dir, exist_ok=True)

    # Representative derivative comparison at N=256
    N = 256
    x = np.linspace(0, 2*np.pi, N, endpoint=False)
    fx = f_true(x)
    df_fft = spectral_derivative_periodic(fx)
    d2f_fft = spectral_second_derivative_periodic(fx)
    df_ex = df_true(x)
    d2f_ex = d2f_true(x)

    plt.figure()
    plt.plot(x, df_ex, label="Analítica f'(x)")
    plt.plot(x, df_fft, '--', label="FFT f'(x)")
    plt.xlabel("x"); plt.ylabel("f'(x)"); plt.legend()
    plt.savefig(os.path.join(fig_dir, "fourier_derivative_comparison.png"),
    ↪  dpi=160, bbox_inches="tight")
    plt.close()

    # Convergence study
    N_list = [16, 32, 64, 128, 256, 512]
    recs = []
    for NN in N_list:
        xx = np.linspace(0, 2*np.pi, NN, endpoint=False)
        fxx = f_true(xx)
        d1 = spectral_derivative_periodic(fxx)
        d2 = spectral_second_derivative_periodic(fxx)
        d1_ex = df_true(xx)
        d2_ex = d2f_true(xx)
        l2_d1 = np.sqrt(np.mean((d1 - d1_ex)**2))
        linf_d1 = np.max(np.abs(d1 - d1_ex))
        l2_d2 = np.sqrt(np.mean((d2 - d2_ex)**2))
        linf_d2 = np.max(np.abs(d2 - d2_ex))
        recs.append(dict(N=NN, L2_err_d1=l2_d1, Linf_err_d1=linf_d1,
        ↪  L2_err_d2=l2_d2, Linf_err_d2=linf_d2))

    import pandas as pd
    df = pd.DataFrame(recs).sort_values("N")
    df.to_csv(os.path.join(tab_dir, "fourier_errors.csv"), index=False)

    plt.figure()
    plt.loglog(df["N"], df["Linf_err_d1"], 'o-', label="|erro| f'")
```

```
    plt.loglog(df["N"], df["L2_err_d1"], 's-', label="|erro|2 f'")
    plt.loglog(df["N"], df["Linf_err_d2"], 'o--', label="|erro| f''")
    plt.loglog(df["N"], df["L2_err_d2"], 's--', label="|erro|2 f''")
    plt.xlabel("N (pontos)"); plt.ylabel("Erro"); plt.legend()
    plt.savefig(os.path.join(fig_dir, "fourier_error_convergence.png"), dpi=160,
    ↪  bbox_inches="tight")
    plt.close()

    # Aliasing demo
    def f_high(x):
        return np.sin(12*x)
    N_small = 16
    x_small = np.linspace(0, 2*np.pi, N_small, endpoint=False)
    f_small = f_high(x_small)
    x_dense = np.linspace(0, 2*np.pi, 2000, endpoint=False)
    # Use FFT-based interpolant from under-sampled data
    c = np.fft.fft(f_small) / N_small
    k = np.fft.fftfreq(N_small, d=1.0/N_small)
    phase = np.outer(k, x_dense)
    f_rec = (c[:, None] * np.exp(1j * phase)).sum(axis=0).real

    plt.figure()
    plt.plot(x_dense, f_high(x_dense), label="Função verdadeira sin(12x)")
    plt.plot(x_dense, f_rec, '--', label=f"Reconstrução com N={N_small}
    ↪  (aliased)")
    plt.scatter(x_small, f_small, s=12, label="Amostras N=16")
    plt.xlabel("x"); plt.ylabel("f(x)"); plt.legend()
    plt.savefig(os.path.join(fig_dir, "fourier_aliasing_demo.png"), dpi=160,
    ↪  bbox_inches="tight")
    plt.close()
```

### A.3.5 legendre_orthogonality.py

```
"""
Legendre orthogonality verification (PCS5029 - Aula 06 - Exercício 5)
Requirements: numpy, matplotlib
Notes:
- Quadrature uses numpy.polynomial.legendre.leggauss(N) with N=400.
- Saves figures under /figures and tables under /tables.
"""

import os
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial.legendre import Legendre, leggauss
import pandas as pd

# Parameters
N_QUAD = 400
MAX_N_FOR_MATRIX = 10
MAX_N_FOR_PLOT = 5
```

```
BASE_DIR = "/mnt/data"
FIG_DIR = os.path.join(BASE_DIR, "figures")
TAB_DIR = os.path.join(BASE_DIR, "tables")
os.makedirs(FIG_DIR, exist_ok=True)
os.makedirs(TAB_DIR, exist_ok=True)

def legendre_basis(n: int) -> Legendre:
    return Legendre.basis(n)

def inner_product_matrix(n_max: int, Nq: int) -> np.ndarray:
    xi, wi = leggauss(Nq)
    Pvals = np.zeros((n_max+1, xi.size))
    for n in range(n_max+1):
        Pvals[n, :] = legendre_basis(n)(xi)
    G = np.zeros((n_max+1, n_max+1))
    for m in range(n_max+1):
        for n in range(n_max+1):
            G[m, n] = np.sum(wi * Pvals[m, :] * Pvals[n, :])
    return G

def main():
    # Compute Gram matrix
    G = inner_product_matrix(MAX_N_FOR_MATRIX, N_QUAD)
    theoretical_diag = np.array([2.0/(2*n+1) for n in
    ↪  range(MAX_N_FOR_MATRIX+1)])

    # Save tables
    G_df = pd.DataFrame(G, index=[f"P{m}" for m in range(MAX_N_FOR_MATRIX+1)],
                           columns=[f"P{n}" for n in range(MAX_N_FOR_MATRIX+1)])
    G_df.to_csv(os.path.join(TAB_DIR, "legendre_orthogonality_matrix.csv"),
    ↪  float_format="%.12e")

    G_err_df = G_df.copy()
    for n in range(MAX_N_FOR_MATRIX+1):
        for m in range(MAX_N_FOR_MATRIX+1):
            target = 0.0 if m != n else theoretical_diag[n]
            G_err_df.iloc[m, n] = G_df.iloc[m, n] - target
    G_err_df.to_csv(os.path.join(TAB_DIR,
    ↪  "legendre_orthogonality_error_matrix.csv"), float_format="%.12e")

    pd.DataFrame({
        "n": np.arange(MAX_N_FOR_MATRIX+1),
        "theoretical_2_over_2n_plus_1": theoretical_diag,
        "numerical_diag": np.diag(G),
        "abs_error": np.abs(np.diag(G) - theoretical_diag)
    }).to_csv(os.path.join(TAB_DIR, "legendre_norms_comparison.csv"),
              index=False, float_format="%.12e")

    # Plot P0..P5
    x_plot = np.linspace(-1, 1, 1000)
    plt.figure()
    for n in range(MAX_N_FOR_PLOT+1):
```

```
        plt.plot(x_plot, legendre_basis(n)(x_plot), label=fr"$P_{5}(x)$")
    plt.xlabel("$x$")
    plt.ylabel("$P_n(x)$")
    plt.title("Legendre Polynomials $P_0$ to $P_5$")
    plt.legend(loc="best", ncol=2, frameon=True)
    plt.grid(True, which="both", linestyle=":")
    plt.savefig(os.path.join(FIG_DIR, "legendre_polynomials.png"), dpi=200,
    ↪  bbox_inches="tight")
    plt.close()

    # Heatmap of G
    plt.figure()
    im = plt.imshow(G, origin="lower", extent=[-0.5, MAX_N_FOR_MATRIX+0.5, -0.5,
    ↪  MAX_N_FOR_MATRIX+0.5])
    plt.colorbar(im, label=r"$\int_{-1}^{1} P_m(x)P_n(x)\,dx$")
    plt.xticks(range(0, MAX_N_FOR_MATRIX+1), [f"P{n}" for n in
    ↪  range(MAX_N_FOR_MATRIX+1)], rotation=45)
    plt.yticks(range(0, MAX_N_FOR_MATRIX+1), [f"P{m}" for m in
    ↪  range(MAX_N_FOR_MATRIX+1)])
    plt.title("Orthogonality Matrix for Legendre Polynomials")
    plt.savefig(os.path.join(FIG_DIR, "legendre_orthogonality_matrix.png"),
    ↪  dpi=200, bbox_inches="tight")
    plt.close()

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

### A.3.6 exercicio6_bvp_chebyshev.py

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

BASE = "/mnt/data"
FIG_DIR = os.path.join(BASE, "figures")
TAB_DIR = os.path.join(BASE, "tables")

os.makedirs(FIG_DIR, exist_ok=True)
os.makedirs(TAB_DIR, exist_ok=True)

def cheb(n: int):
    if n == 0:
        return 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) ** np.arange(n + 1))
    X = np.tile(x, (n + 1, 1)); dX = X - X.T
    D = (np.outer(c, 1 / c)) / (dX + np.eye(n + 1))
    D = D - np.diag(np.sum(D, axis=1))
```

```
    return D, x

pi = np.pi
n = 31
N = n + 1

D_raw, x = cheb(n)
D1 = -D_raw              # sign correction so D1 acts like d/dx
D2 = D1 @ D1
I = np.eye(N)

f = -pi * np.sin(pi * x)
y_true = np.cos(pi * x)
L = D2 + D1 + (pi**2) * I

def impose_dirichlet(Lmat, rhs):
    A = Lmat.copy(); b = rhs.copy()
    A[-1,:] = 0.0; A[-1,-1] = 1.0; b[-1] = -1.0
    A[0,:]  = 0.0; A[0,0]  = 1.0;  b[0]  = -1.0
    return A, b

def impose_neumann(Lmat, rhs, D1):
    A = Lmat.copy(); b = rhs.copy()
    A[-1,:] = D1[-1,:]; b[-1] = 0.0
    A[0,:]  = D1[0,:];  b[0]  = 0.0
    return A, b

def impose_robin(Lmat, rhs, D1, alpha=1.0, beta=1.0, g=-1.0):
    A = Lmat.copy(); b = rhs.copy()
    A[-1,:] = alpha*np.eye(N)[-1,:] + beta*D1[-1,:]; b[-1] = g
    A[0,:]  = alpha*np.eye(N)[0,:]  + beta*D1[0,:];  b[0]  = g
    return A, b

# Solve
A_dir,b_dir = impose_dirichlet(L, f)
y_dir = np.linalg.solve(A_dir, b_dir)

A_neu,b_neu = impose_neumann(L, f, D1)
y_neu = np.linalg.solve(A_neu, b_neu)

A_rob,b_rob = impose_robin(L, f, D1, 1.0, 1.0, -1.0)
y_rob = np.linalg.solve(A_rob, b_rob)

# Errors
err_dir = float(np.max(np.abs(y_dir - y_true)))
err_neu = float(np.max(np.abs(y_neu - y_true)))
err_rob = float(np.max(np.abs(y_rob - y_true)))

# Plots
sort_idx = np.argsort(x)
xs = x[sort_idx]
```

```
def save_plot(path, y_num, title):
    plt.figure()
    plt.plot(xs, y_true[sort_idx], linestyle="--", label="Analítica: cos(pi x)")
    plt.plot(xs, y_num[sort_idx], label="Numérica")
    plt.xlabel("x"); plt.ylabel("y(x)")
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    plt.savefig(path, dpi=150)
    plt.close()

save_plot(os.path.join(FIG_DIR, "y_dirichlet.png"), y_dir, "BVP com CC
↪  Dirichlet")
save_plot(os.path.join(FIG_DIR, "y_neumann.png"), y_neu, "BVP com CC Neumann")
save_plot(os.path.join(FIG_DIR, "y_robin.png"), y_rob, "BVP com CC Robin")

plt.figure()
plt.plot(xs, y_dir[sort_idx], label="Dirichlet")
plt.plot(xs, y_neu[sort_idx], label="Neumann")
plt.plot(xs, y_rob[sort_idx], label="Robin")
plt.xlabel("x"); plt.ylabel("y(x)")
plt.title("Comparação das soluções numéricas (Dirichlet, Neumann, Robin)")
plt.legend()
plt.tight_layout()
plt.savefig(os.path.join(FIG_DIR, "y_comparison.png"), dpi=150)
plt.close()

# Table (CSV)
errors_df = pd.DataFrame({
    "Condicao_de_Contorno": ["Dirichlet", "Neumann", "Robin"],
    "n": [n, n, n],
    "Max_erro_infinito": [err_dir, err_neu, err_rob]
})
errors_df.to_csv(os.path.join(TAB_DIR, "ex6_max_errors.csv"), index=False)

print("Done. Figures in", FIG_DIR, "Tables in", TAB_DIR, "Errors:", err_dir,
↪  err_neu, err_rob)
```

—

## A.4 A.4 Observações finais sobre o ambiente

- Todos os scripts foram testados em ambiente `Python 3.12.10` sob Linux e WSL2;
- As figuras foram geradas automaticamente nas pastas `/figures` e `/tables`;
- As execuções típicas consomem poucos segundos e menos de 1 GB de memória;
- As bibliotecas utilizadas são totalmente open source.

**Nota:** Este apêndice é parte integrante do relatório, servindo para garantir a reprodutibilidade dos resultados apresentados nas seções principais.

# B Declaração de uso de LLM

Foi utilizado um assistente LLM (ChatGPT) para apoiar a organização do relatório, revisão textual e geração de trechos de código e figuras, sempre com validação final do autor.