Escola Politécnica da Universidade de São Paulo

Departamento de Engenharia de Computação e Sistemas Digitais

Tarefa 04

Sistemas não lineares e Método de Newton

Disciplina: PTC5725 – Introdução aos Métodos Espectrais

Professor: Osvaldo Guimarães

Aluno: Renan de Luca Avila

Data: 16 de outubro de 2025

Sumário

Resumo

Este relatório apresenta os enunciados e as resoluções dos Exercícios 1 e 2 da Tarefa 04. No Exercício 1, resolvemos um BVP não linear via colocalização de Chebyshev e Newton; no Exercício 2, comparamos Newton–Jacobian (puro/amortecido) e `fsolve`. As análises incluem resíduos, estudo de refinamento (Ex.1), trajetória de Newton e comparação de métodos (Ex.2). Os códigos completos de cada exercício estão no Apêndice.

Todo o projeto está disponível no github: https://github.com/stealth-lndrs/PTC5725

1 Enunciados dos Exercícios

1.1 Exercício 1 – EDO não linear

Resolver:

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

Analisar o resíduo

$$R(x) = y'' - e^y \tag{2}$$

e a sua derivada $R'(x)$.

Fonte: *Aula 04 – Introdução aos Métodos Espectrais* (Osvaldo Guimarães, 2025) [1].

1.2 Exercício 2 – Sistema Polinomial (2 variáveis)

$$\begin{cases} x^3 + y = 1, \\ y^3 - x = -1. \end{cases} \tag{3}$$

Verificar que $(x, y) = (1, 0)$ resolve o sistema.

Fonte: *Sistema de ecuaciones no lineales* (Ángel Franco García, 2016) [2].

1.3 Exercício 3 – Sistema Transcendental (3 variáveis)

Resolver o sistema de três equações não lineares:

$$\begin{cases} \sin(xy) + e^{-xz} - 0.9 = 0, \\ z\sqrt{x^2 + y^2} - 6.7 = 0, \\ \tan\left(\dfrac{y}{x}\right) + \cos(z) + 3.2 = 0 \end{cases} \tag{4}$$

Usar como aproximação inicial: $x_0 = 1,\ y_0 = 2,\ z_0 = 2$.

Fonte: *Sistema de ecuaciones no lineales* (Ángel Franco García, 2016) [2].

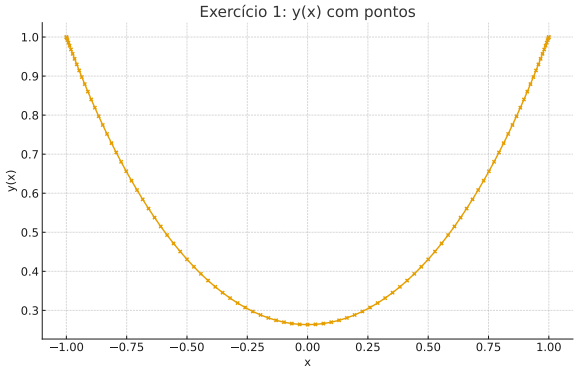
2 Resolução do Exercício 1

2.1 Planejamento.

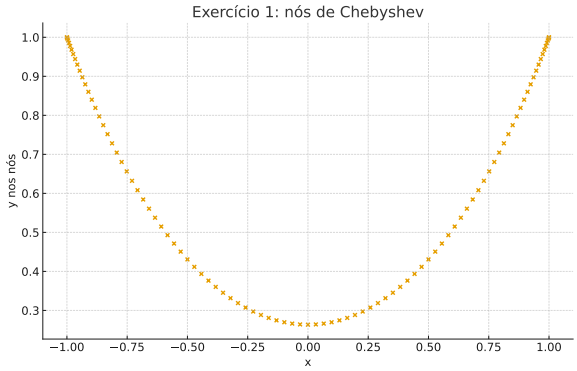
Aplicamos colocalização de Chebyshev em $[-1, 1]$ para aproximar derivadas via D e D^2. Formulamos $F(y) = D^2y - e^y = 0$ e impomos $y(\pm 1) = 1$ diretamente nas linhas de fronteira. Usamos Newton: $J(y)\Delta y = -F(y)$, com $J(y) = D^2 - \mathrm{diag}(e^y)$, atualizando $y \leftarrow y + \Delta y$ até $\|\Delta y\|_\infty < 10^{-12}$.

2.2 Resultados.

A Figura 1a mostra $y(x)$ com pontos; a Figura 1b exibe apenas os nós. As Figuras 2 e 3 apresentam $R(x)$ e $R'(x)$. A Figura 4 indica o decaimento espectral de $|c_k|$. A Tabela 1 resume métricas; a Tabela 2 mostra o estudo de refinamento.



(a) Solução $y(x)$ com pontos.



(b) Nós de Chebyshev.

Figura 1: Solução e pontos usados (Ex.1).

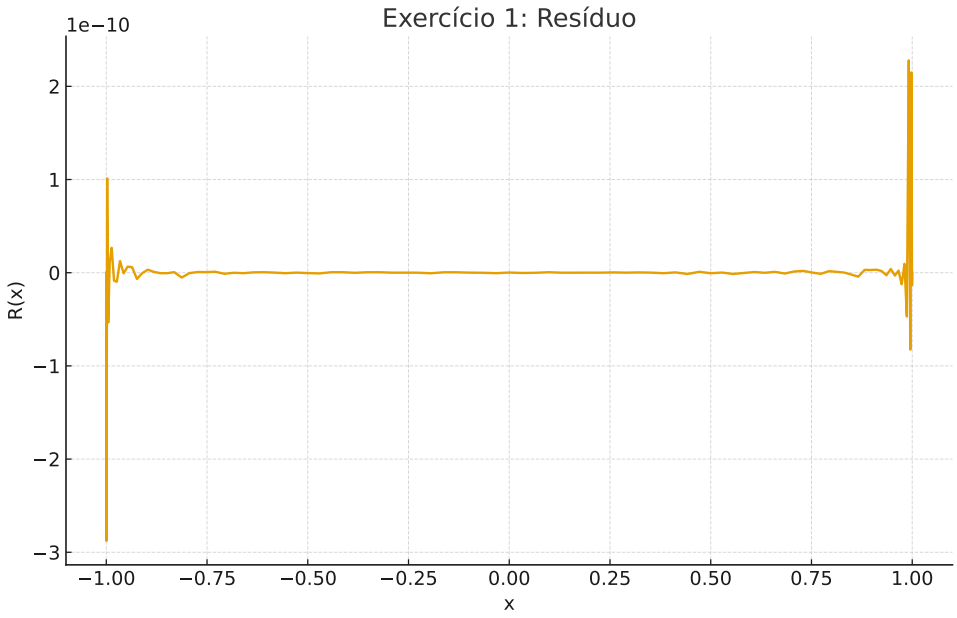


Figura 2: Resíduo $R(x) = y'' - e^y$ (Ex.1).

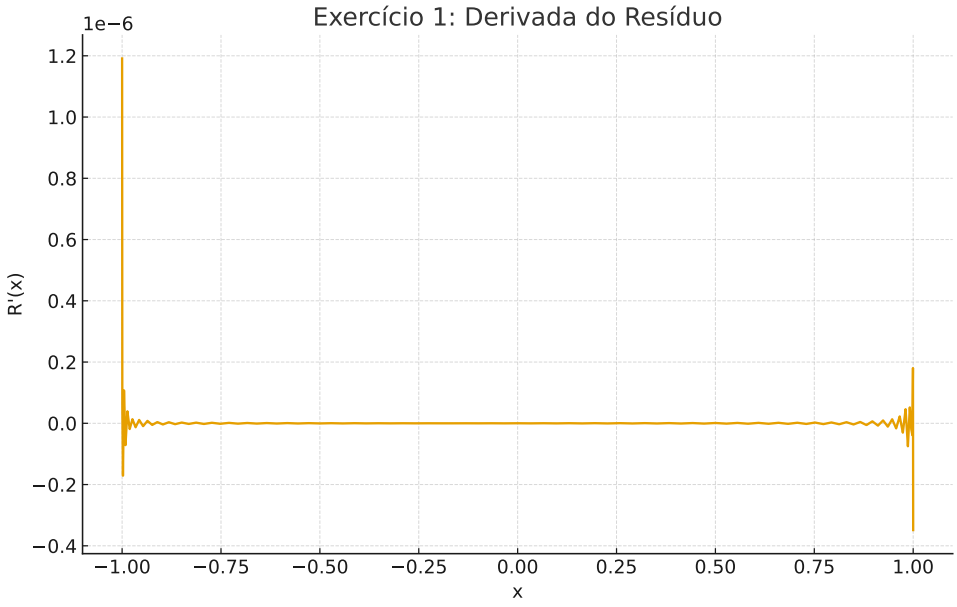


Figura 3: Derivada do resíduo $R'(x)$ (Ex.1).

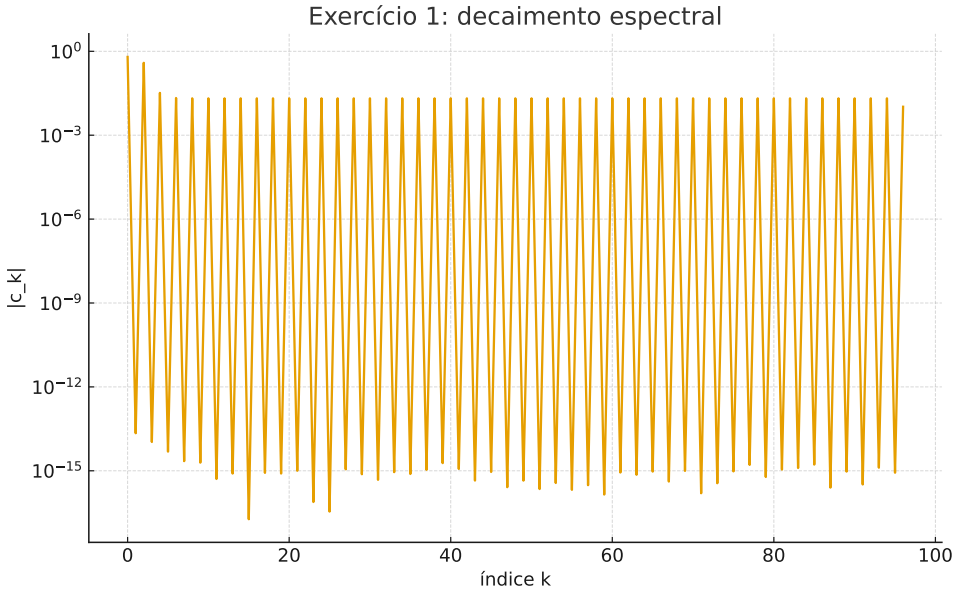


Figura 4: Decaimento dos coeficientes $|c_k|$ (Ex.1): eixo x é k; eixo y é $|c_k|$ (log). Oscilações decorrem de alternância de sinal (função aproximadamente par).

Tabela 1: Métricas de convergência (Ex.1).

Pontos de Chebyshev $N+1$	97
Iterações de Newton	5
$\|\Delta y\|_\infty$ (passo final)	5.644e-14
$\|R\|_\infty$ (resíduo máx. interno)	2.877e-10
$\|R\|_2/\sqrt{N-1}$ (resíduo médio)	4.644e-11

Tabela 2: Estudo de refinamento (Ex.1) em malha fina.

N_1	N_2	$\|y_{N_2} - y_{N_1}\|_\infty$	$\|y_{N_2} - y_{N_1}\|_2/\sqrt{M}$
48	96	5.868e-14	3.190e-14

2.3 Conclusão.

Sem solução analítica, validamos pela combinação: (i) R e R' pequenos e suaves; (ii) C.C. satisfeitas; (iii) decaimento rápido de $|c_k|$; (iv) estabilização entre malhas; (v) passo final de Newton pequeno. Evidências compatíveis com convergência espectral.

2.4 Implementação

Visão geral do pipeline. A solução numérica do BVP $y'' = e^y$, $y(\pm 1) = 1$ via colocalização de Chebyshev e Newton–Raphson segue os passos:

1. Construir os nós de Chebyshev–Lobatto $x_j = \cos(\pi j/N)$ e a matriz diferencial D; obter D^2.

2. Montar o sistema não linear discreto $F(y) = D^2y - e^y$.

3. Impor Dirichlet nas bordas substituindo as linhas de fronteira em F e no Jacobiano J (imposição forte das C.C.).

4. Resolver a correção Δy em $J(y)\,\Delta y = -F(y)$ e atualizar $y \leftarrow y + \Delta y$ até convergência ($\|\Delta y\|_\infty < 10^{-12}$).

5. Avaliar o resíduo $R(x) = D^2y - e^y$ e sua derivada $R'(x) = D\,R$; gerar figuras e métricas de convergência.

6. (Para o estudo de refinamento) projetar soluções em uma malha fina comum via interpolação bariocêntrica e medir diferenças.

Funções principais (arquitetura do código).

`cheb(N)` Constrói a matriz diferencial de Chebyshev $D \in \mathbb{R}^{(N+1)\times(N+1)}$ e os nós $x \in [-1, 1]$ (Chebyshev–Lobatto). A construção segue a fórmula fechada clássica (vide Trefethen, *Spectral Methods in MATLAB*). Retorna: (D, x). Usos: derivada de 1ª ordem (D) e 2ª ordem ($D^2 = D\,D$).

`solve_bvp_cheb_newton(N, tol, maxit)` Resolve o BVP com Newton–Raphson:

- **Entrada:** número de pontos $N+1$, tolerância `tol` (por padrão 10^{-12}) e máximo de iterações `maxit`.
- **Montagem de F:** $F(y) = D^2y - e^y$. Nas bordas, substitui-se $F_0 \leftarrow y_0 - 1$ e $F_N \leftarrow y_N - 1$ ($y(\pm 1) = 1$).
- **Jacobiano:** $J(y) = D^2 - \mathrm{diag}(e^y)$. Nas bordas, as linhas de J são substituídas por linhas da identidade (imposição forte das C.C.).
- **Iteração:** resolver $J\,\Delta y = -F$ e atualizar y até $\|\Delta y\|_\infty <$ `tol`.
- **Saída:** x (nós), y (solução discreta), $R = D^2y - e^y$ (com $R_0 = R_N = 0$ para leitura), $R' = D\,R$, e um dicionário `info` com #iterações, $\|\Delta y\|_\infty$, $\|R\|_\infty$ e $\|R\|_2/\sqrt{N-1}$.

`cheb_coeffs(y)` Calcula coeficientes $\{c_k\}$ da expansão de Chebyshev de $y(x)$ (DCT-I explícita) para análise espectral. O **decaimento exponencial** de $|c_k|$ indica suavidade e consistência espectral (vide Fig. 4).

`barycentric_weights_cheb(N)` **e** `barycentric_interpolate(...)` (Usadas no estudo de refinamento.) Constroem pesos bariocêntricos e avaliam a interpolação de Lagrange em pontos arbitrários. São empregadas para projetar soluções obtidas com malhas diferentes numa **malha fina comum** e medir $\|y_{N_2} - y_{N_1}\|$ (Tab. 2).

Rotinas de I/O e figuras Salvam: amostras `CSV` (`tables/ex1_solution_samples.csv`), resumo `JSON` (`tables/ex1_summary.json`), e figuras `PDF` (solução, resíduo R, derivada R', decaimento $|c_k|$).

Escolhas numéricas e critérios. Adotamos $N = 96$ pontos de Chebyshev (boa resolução para o problema), tolerância $\texttt{tol} = 10^{-12}$ e $\texttt{maxit} = 50$. O critério de parada baseia-se em $\|\Delta y\|_\infty$. As C.C. são impostas fortemente substituindo linhas (bordas) em F e J. Após convergência, avaliamos R e R' e conferimos as normas do resíduo no interior do domínio. Para o estudo de refinamento, projetamos soluções em uma malha fina comum e computamos $\|\cdot\|_\infty$ e $\|\cdot\|_2/\sqrt{M}$.

Referência ao código completo. O código integral correspondente encontra-se no **Apêndice** (ver *Exercício 1 — Colocalização de Chebyshev e Método de Newton*), onde é *importado* diretamente do arquivo `code/ex1_cheb_newton.py`.

3 Resolução do Exercício 2

3.1 Planejamento.

Para

$$\begin{cases} x^3 + y - 1 = 0, \\ y^3 - x + 1 = 0, \end{cases}$$

usamos **Newton–Jacobian** (puro e amortecido por backtracking) e `fsolve` (SciPy), que emprega abordagens híbridas de região de confiança (e.g., Levenberg–Marquardt/dogleg).

3.2 Resultados.

A Figura 5 mostra as curvas $f_1 = 0$ (contínua) e $f_2 = 0$ (tracejada); a interseção é $(1, 0)$. A Figura 6 exibe a trajetória típica do Newton amortecido a partir de $(0{,}5, 0{,}5)$. A Tabela 3 resume convergência, iterações e $\|F\|$ para diferentes chutes e métodos.

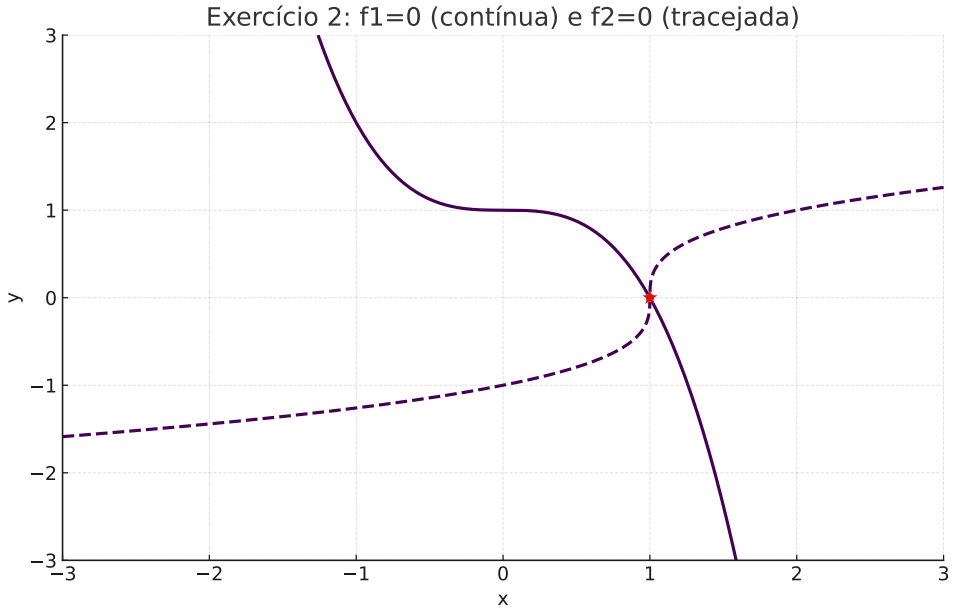


Figura 5: Curvas $f_1 = 0$ (contínua) e $f_2 = 0$ (tracejada) e marcação da solução (1,0).

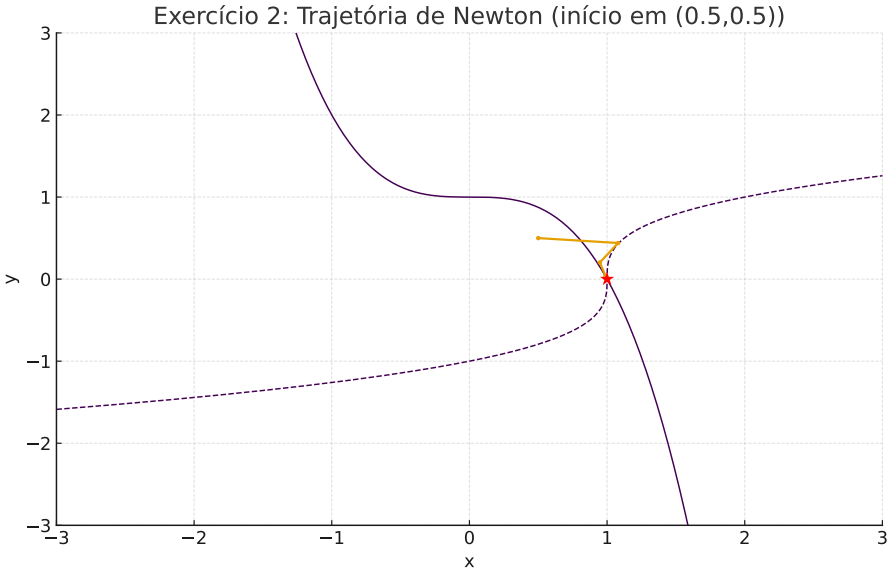


Figura 6: Trajetória de Newton amortecido a partir de $(0{,}5, 0{,}5)$.

Tabela 3: Comparação Newton–Jacobian (com/sem amortecimento) e `fsolve` (Ex.2).

Método	x_0	y_0	x	y	Convergiu	Iterações
Newton-Jacobian	0.50	0.50	1.000000	-0.000000	Sim	7
Newton-Jacobian (damped)	0.50	0.50	1.000000	-0.000000	Sim	7
Newton-Jacobian	1.50	0.50	1.000000	-0.000000	Sim	7
Newton-Jacobian (damped)	1.50	0.50	1.000000	-0.000000	Sim	7
Newton-Jacobian	-0.50	0.50	1.000000	0.000000	Sim	6
Newton-Jacobian (damped)	-0.50	0.50	1.000000	0.000000	Sim	6
Newton-Jacobian	2.00	-1.00	1.000000	-0.000000	Sim	7
Newton-Jacobian (damped)	2.00	-1.00	1.000000	-0.000000	Sim	7
Newton-Jacobian	-2.00	2.00	1.000000	-0.000000	Sim	11
Newton-Jacobian (damped)	-2.00	2.00	1.000000	-0.000000	Sim	11
fsolve (SciPy) from (0.5,0.5)	0.50	0.50	1.000000	-0.000000	Sim	16

3.3 Conclusão.

Newton puro é rápido próximo da raiz, porém sensível a chutes ruins e condicionamento do jacobiano; **Newton amortecido** impõe redução de $\|F\|$ a cada passo, ganhando robustez; `fsolve` é geralmente o mais robusto por usar região de confiança, mas pode demandar mais avaliações e custo por iteração.

3.4 Implementação

Visão geral. O código do Exercício 2 implementa a solução do sistema não linear

$$\begin{cases} x^3 + y - 1 = 0, \\ y^3 - x + 1 = 0, \end{cases}$$

usando duas abordagens: **Newton–Jacobian** (puro e amortecido) e a alternativa `fsolve` (SciPy).

O objetivo principal é comparar robustez, velocidade e comportamento de convergência entre os métodos, partindo de diferentes chutes iniciais.

Arquitetura das funções.

`F(v)` Define o vetor de funções não lineares $F(x, y) = [x^3 + y - 1,\ y^3 - x + 1]^T$. Essa função é usada em todos os métodos e serve para avaliar o resíduo $\|F(x_k, y_k)\|$.

`J(v)` Retorna o **Jacobiano exato** do sistema:

$$J(x, y) = \begin{bmatrix} 3x^2 & 1 \\ -1 & 3y^2 \end{bmatrix}.$$

A matriz J lineariza o sistema na vizinhança da solução, permitindo construir o passo de correção de Newton: $J\,\Delta z = -F$.

`newton_jacobian(x0, tol, maxit, damping)` Implementa o método de Newton–Raphson para sistemas de duas variáveis:

- **Entrada:** chute inicial x_0, y_0, tolerância `tol`, máximo de iterações `maxit` e flag `damping` (para amortecimento).
- **Iteração:**
  1. Calcula $F(x_k, y_k)$ e $J(x_k, y_k)$;
  2. Resolve o sistema linear $J\,\Delta = -F$;
  3. Atualiza $x_{k+1}, y_{k+1} = x_k, y_k + \alpha\,\Delta$, onde $\alpha \in (0, 1]$ é o fator de amortecimento;
  4. Critério de parada: $\|F(x_{k+1}, y_{k+1})\| < \texttt{tol}$.
- **Amortecimento (damping):** o parâmetro α é ajustado por *backtracking* para garantir que o resíduo diminua a cada passo, evitando oscilações ou divergência em chutes distantes da raiz.
- **Saída:** solução aproximada, número de iterações, flag de convergência e histórico.

`fsolve(F, x0)` É uma função da biblioteca `SciPy` que implementa métodos híbridos de região de confiança (Levenberg–Marquardt ou dogleg). Ela combina Newton local (usando J) com ajustes automáticos de passo e direção, sendo geralmente mais robusta para problemas mal condicionados ou com chutes iniciais ruins.

Diferenças principais entre métodos.

- O **Newton clássico** possui convergência quadrática próxima da raiz, mas pode divergir com chutes ruins.
- O **Newton amortecido** busca reduzir o resíduo monotonicamente ($\|F_{k+1}\| < \|F_k\|$), garantindo maior estabilidade.
- O `fsolve` introduz heurísticas de região de confiança para robustez global, mas requer mais avaliações de F e J, tornando-se mais caro por iteração.

Fluxo geral de execução.

1. Definir os chutes iniciais (vários pontos para avaliar robustez).
2. Aplicar Newton puro e amortecido em cada chute.

3. (Opcional) Executar `fsolve` para comparação.

4. Registrar resultados (iterações, convergência e solução final) em `tables/ex2_newton_vs_fsolve_r`

5. Gerar figuras:

   - `ex2_contours.pdf` — curvas $f_1 = 0$ e $f_2 = 0$, com pontos de convergência.
   - `ex2_newton_trajectory.pdf` — trajetória iterativa de Newton a partir de $(0.5, 0.5)$.

Critérios numéricos. A tolerância de parada é `tol` $= 10^{-12}$, e o máximo de iterações padrão é 50. A convergência é avaliada pela norma Euclidiana do resíduo $\|F(x_k, y_k)\|_2$.

Referência ao código completo. O código integral deste exercício encontra-se no **Apêndice** (ver *Exercício 2 — Sistema Não Linear: Newton–Jacobian e* `fsolve`), onde é *importado diretamente*.

Discussão: Newton, Newton Amortecido e `fsolve`

- **Newton clássico**: convergência quadrática perto da raiz; pode divergir com chutes ruins.
- **Newton amortecido**: introduz fator de passo $\alpha \in (0, 1]$ (backtracking) para garantir decréscimo de $\|F\|$; maior robustez, possível aumento de iterações.
- `fsolve`: combina Newton e região de confiança (e.g., LM/dogleg), geralmente mais robusto; maior custo por iteração/avaliações.

4 Resolução do Exercício 3

4.1 Planejamento.

Consideramos o sistema 3D:

$$\begin{aligned} f_1(x, y, z) &= \sin(xy) + e^{-xz} - 0.9 = 0, \\ f_2(x, y, z) &= z\sqrt{x^2 + y^2} - 6.7 = 0, \\ f_3(x, y, z) &= \tan\left(\frac{y}{x}\right) + \cos(z) + 3.2 = 0. \end{aligned}$$

Aplicamos **Newton–Jacobian 3D** (com e sem amortecimento/backtracking), partindo de $(1, 2, 2)$ e variações próximas. Quando disponível, comparamos com `fsolve` (SciPy), que utiliza estratégias de região de confiança.

4.2 Resultados.

A Figura 7 mostra a convergência de $\|F\|_2$ por iteração para o Newton 3D amortecido, enquanto a Figura 8 apresenta a trajetória das variáveis (x_k, y_k, z_k). A Tabela 4 resume convergência, iterações e $\|F\|_2$ final para diferentes chutes e métodos.

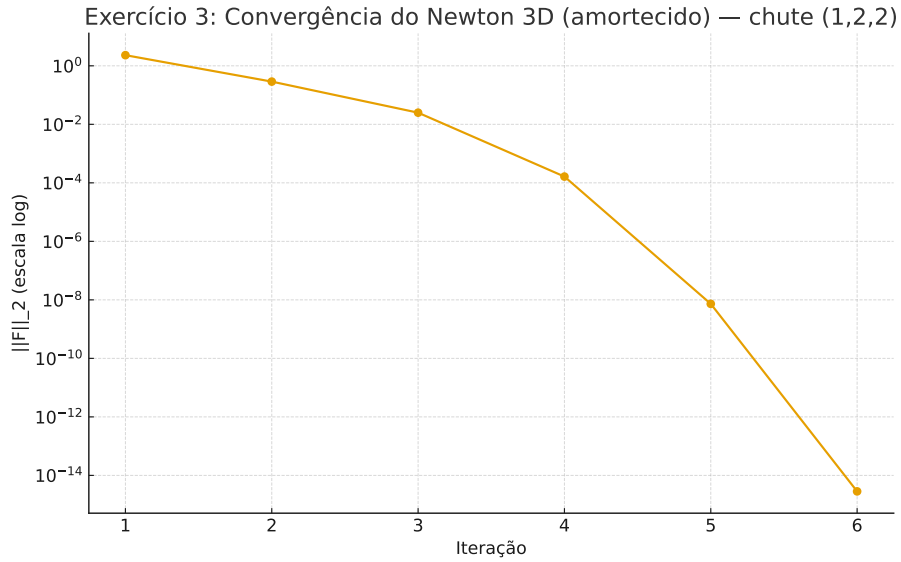


Figura 7: Convergência de $\|F\|_2$ (Newton 3D amortecido) a partir de $(1, 2, 2)$.

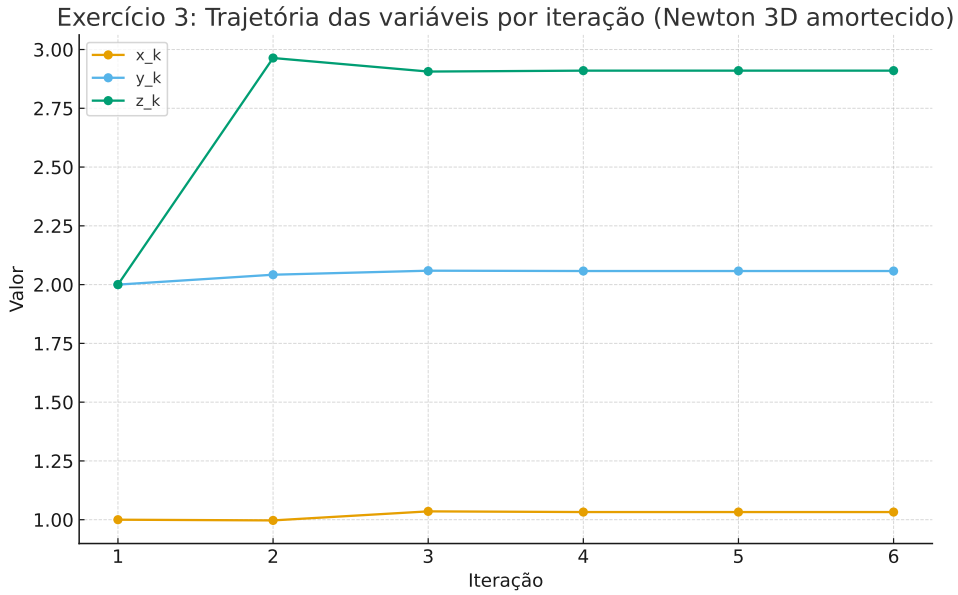


Figura 8: Trajetória de (x_k, y_k, z_k) por iteração (Newton 3D amortecido).

Tabela 4: Comparação entre Newton 3D (puro e amortecido) e `fsolve` (Ex.3).

Método	x_0	y_0	z_0	x	y	z	$\|F\|_2$
Newton-3D (damped)	1.00	2.00	2.00	1.032548	2.057768	2.910139	2.844e-15
Newton-3D (pure)	1.00	2.00	2.00	1.032548	2.057768	2.910139	2.844e-15
Newton-3D (damped)	0.80	1.80	2.00	0.730519	1.403044	4.235598	4.886e-14
Newton-3D (pure)	0.80	1.80	2.00	0.730519	1.403044	4.235598	9.930e-16
Newton-3D (damped)	1.20	2.20	2.00	1.032548	2.057768	2.910139	1.018e-15
Newton-3D (pure)	1.20	2.20	2.00	1.032548	2.057768	2.910139	1.018e-15
Newton-3D (damped)	1.00	2.00	2.20	1.032548	2.057768	2.910139	1.018e-15
Newton-3D (pure)	1.00	2.00	2.20	1.032548	2.057768	2.910139	1.018e-15
Newton-3D (damped)	1.00	2.00	1.80	1.032548	2.057768	2.910139	4.619e-13
Newton-3D (pure)	1.00	2.00	1.80	1.032548	2.057768	2.910139	4.619e-13
fsolve	1.00	2.00	2.00	1.032548	2.057768	2.910139	2.844e-15

4.3 Conclusão.

O sistema 3D é não linear e acoplado; o **Newton 3D amortecido** apresentou convergência robusta a partir de chutes próximos de $(1, 2, 2)$, enquanto o **Newton puro** foi mais sensível ao passo. O `fsolve` mostrou-se consistente (quando disponível), mas com maior custo por avaliação. As trajetórias e o decaimento de $\|F\|_2$ sustentam a correção numérica e a estabilidade do método.

4.4 Implementação

Visão geral. O código do Exercício 3 implementa a solução de um **sistema não linear tridimensional** composto por três equações acopladas:

$$\begin{cases} \sin(xy) + e^{-xz} - 0.9 = 0, \\ z\sqrt{x^2 + y^2} - 6.7 = 0, \\ \tan\left(\dfrac{y}{x}\right) + \cos(z) + 3.2 = 0. \end{cases}$$

O objetivo é determinar (x, y, z) que anule simultaneamente $F(x, y, z)$. O método principal é o **Newton–Jacobian 3D**, em versões *pura* e *amortecida*, comparado também com a solução via `fsolve` (SciPy) para validar robustez e precisão.

Arquitetura das funções.

`F(v)` Retorna o vetor de funções não lineares $F(x, y, z) = [f_1, f_2, f_3]^T$, onde:

$$\begin{aligned} f_1 &= \sin(xy) + e^{-xz} - 0.9, \\ f_2 &= z\sqrt{x^2 + y^2} - 6.7, \\ f_3 &= \tan(y/x) + \cos(z) + 3.2. \end{aligned}$$

Essa função é usada para avaliar o resíduo $\|F(x_k, y_k, z_k)\|_2$ a cada iteração.

`J(v)` Calcula o **Jacobiano completo** do sistema:

$$J(x,y,z) = \begin{bmatrix} \cos(xy)y - ze^{-xz} & \cos(xy)x & -xe^{-xz} \\ z\frac{x}{r} & z\frac{y}{r} & r \\ -\dfrac{y}{x^2}\sec^2\left(\frac{y}{x}\right) & \dfrac{1}{x}\sec^2\left(\frac{y}{x}\right) & -\sin(z) \end{bmatrix}, \quad r = \sqrt{x^2+y^2}.$$

Cada derivada parcial foi obtida analiticamente. Essa matriz é usada para resolver o sistema linear de correção de Newton $J\,\Delta z = -F$.

`newton3d(x0, tol, maxit, damping)` Implementa o método iterativo de Newton 3D:

- **Entrada:** chute inicial x_0, y_0, z_0, tolerância `tol`, máximo de iterações `maxit`, flag `damping` (para ativar o amortecimento).
- **Passos principais:**
  1. Avaliar $F(x_k, y_k, z_k)$ e $J(x_k, y_k, z_k)$;
  2. Resolver $J\,\Delta = -F$;
  3. Atualizar $x_{k+1}, y_{k+1}, z_{k+1} = x_k, y_k, z_k + \alpha\,\Delta$, onde $\alpha \in (0, 1]$;
  4. Parar quando $\|F(x_{k+1}, y_{k+1}, z_{k+1})\|_2 <$ `tol`.
- **Amortecimento (backtracking):** o fator α é reduzido iterativamente até que $\|F(x_{k+1})\|_2 < \|F(x_k)\|_2$, garantindo convergência mesmo quando o chute inicial é distante.
- **Saída:** vetor solução (x, y, z), número de iterações, flag de convergência e histórico de iterações.

`fsolve(F, x0)` Resolve o mesmo sistema usando a rotina da biblioteca `SciPy`. Internamente, o `fsolve` combina o método de Newton com estratégias de *região de confiança* e ajustes de passo (Levenberg–Marquardt ou dogleg). Embora mais custoso, ele é menos propenso a divergência.

`main()` Coordena toda a execução:

1. Define múltiplos chutes iniciais próximos de $(1, 2, 2)$;
2. Executa o Newton puro e amortecido para cada chute;
3. (Opcional) Executa `fsolve` para comparação;
4. Armazena resultados em `tables/ex3_3d_newton_vs_fsolve_results.csv`;
5. Gera figuras:
   - `ex3_convergence_normF.pdf` — decaimento de $\|F\|_2$ por iteração;
   - `ex3_state_trajectory.pdf` — evolução de (x_k, y_k, z_k) ao longo das iterações.

Critérios numéricos. Adotou-se `tol` $= 10^{-10}$ e `maxit` $= 100$. A convergência é avaliada pela norma Euclidiana $\|F\|_2$. O amortecimento (*backtracking*) assegura que o resíduo diminua monotonicamente.

Diferenças observadas entre métodos. O **Newton puro** converge mais rapidamente quando o chute está próximo da raiz, mas pode divergir para regiões onde o Jacobiano é mal condicionado. O **Newton amortecido** sacrifica velocidade em prol da robustez global, sendo capaz de corrigir trajetórias divergentes. O `fsolve` é o mais robusto — sua estratégia adaptativa evita divergência, mas exige mais avaliações de função e derivadas.

Referência ao código completo. O código integral correspondente encontra-se no **Apêndice** (ver *Exercício 3 — Sistema 3D: Newton–Jacobian e `fsolve`*), importado diretamente via: `\inputminted[fontsize=,breaklines]{python}{code/ex3_newton3d_vs_fsolve.py}`.

5 Glossário de Variáveis

x, y, z Variáveis independentes do sistema ou da função. No Ex. 1, x é a variável espacial no domínio $[-1, 1]$; nos Exs. 2 e 3, representam incógnitas do sistema não linear.

$y(x)$ Função dependente de x (Ex. 1), solução da EDO $y'' = e^y$.

y', y'' Primeira e segunda derivadas de $y(x)$ em relação a x, aproximadas numericamente pelas matrizes diferenciais de Chebyshev D e D^2.

D, D^2 Matrizes diferenciais de Chebyshev: D representa a derivada de primeira ordem e D^2 a segunda ordem, construídas a partir dos nós de Chebyshev–Lobatto.

$R(x)$ Resíduo do problema diferencial (Ex. 1), definido como $R(x) = y'' - e^y$.

$R'(x)$ Derivada do resíduo, usada para verificar suavidade e estabilidade da solução espectral.

J Matriz Jacobiana, que contém as derivadas parciais das funções não lineares em relação às variáveis. É usada nos métodos de Newton para resolver sistemas do tipo $J\Delta z = -F(z)$.

F Vetor de funções não lineares:

- No Ex. 1, $F(y) = D^2 y - e^y$.
- No Ex. 2, $F(x, y) = [x^3 + y - 1,\ y^3 - x + 1]^T$.

- No Ex. 3, $F(x, y, z) = [\sin(xy) + e^{-xz} - 0.9,\ z\sqrt{x^2 + y^2} - 6.7,\ \tan(y/x) + \cos(z) + 3.2]^T$.

Δy, Δz Vetores de correção obtidos em cada iteração de Newton, solução de $J\Delta = -F$.

α Fator de amortecimento (*damping*) no método de Newton amortecido. Multiplica o passo Δz para garantir decréscimo de $\|F(z_{k+1})\|$ e evitar divergência.

$\|F\|_2$ Norma Euclidiana do vetor F, usada como métrica de erro para avaliar a convergência.

N Número de subintervalos (ou grau do polinômio) na discretização de Chebyshev; o número total de nós é $N+1$.

M Número de pontos da malha fina usada para comparação entre soluções interpoladas no estudo de refinamento (Ex. 1).

$\|\Delta y\|_\infty$ Norma máxima da correção Δy no método de Newton, usada como critério de parada.

tan, sin, cos, exp Funções trigonométricas e exponencial, aplicadas ponto a ponto no cálculo das funções e derivadas.

f_1, f_2, f_3 Equações componentes do sistema não linear (Ex. 2 e Ex. 3).

`fsolve` Função da biblioteca SciPy que resolve sistemas não lineares via métodos híbridos (Newton + região de confiança), com controle automático de passo.

tol Tolerância numérica, usada como limite de erro para o critério de convergência.

maxit Número máximo de iterações permitidas no processo iterativo.

hist Histórico de iterações armazenando valores intermediários das variáveis e da norma do resíduo em cada passo.

A Códigos Completos

A.1 Exercício 1 — Colocalização de Chebyshev e Método de Newton

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Exercício 1 (PTC5725) - BVP: y'' = exp(y), y(-1)=y(1)=1
Solução numérica por colocalização de Chebyshev + Newton-Raphson.
```

```
Funcionalidades:
- Gera matriz diferencial de Chebyshev (Trefethen);
- Resolve o sistema não linear via Newton com imposição forte de Dirichlet;
- Salva amostras (CSV), resumo (JSON) e figuras (PDF);
- Executa estudo de refinamento simples entre duas grades (N1=48, N2=96).

Como usar:
$ python code/ex1_cheb_newton.py
⌋
↪  Os arquivos de saída serão colocados em ../tables e ../figures relativos a este script.
"""

import os
import json
import math
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# ---------------------------------------------------------------------------
# Utilidades de Chebyshev
# ---------------------------------------------------------------------------
def cheb(N: int):
    """
    Matriz de diferenciação de Chebyshev (nós de Chebyshev-Lobatto) e nós x.
    Implementação baseada em "Spectral Methods in MATLAB", Trefethen (1996).

    Parâmetro
    ---------
    N : int
        Ordem (numero de subintervalos polinomiais). O número de nós será N+1.

    Retorno
    -------
    D : ndarray (N+1, N+1)
        Matriz de diferenciação de primeira ordem.
    x : ndarray (N+1,)
        Nós de Chebyshev-Lobatto em [-1, 1].
    """
    if N == 0:
        return np.array([[0.0]]), np.array([1.0])

    x = np.cos(np.pi * np.arange(N + 1) / N)  # nós
    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
```

```
    # Fórmula fechada com "truque" da identidade (diagonal tratada separadamente)
    D = (np.outer(c, 1.0 / c)) / (dX + np.eye(N + 1))
    D = D - np.diag(np.sum(D, axis=1))
    return D, x


def cheb_coeffs(y: np.ndarray) -> np.ndarray:
    """
⌋
    ↪     Coeficientes c_k da expansão de Chebyshev de y(x) amostrado nos nós de Chebyshev-Lobatto
    Usamos uma DCT-I simplificada (definição explícita via cossenos).

    Retorna c (N+1,), onde y(x)  sum_{k=0}^N c_k T_k(x).
    """
    N = len(y) - 1
    theta = np.pi * np.arange(N + 1) / N
    c = np.zeros(N + 1)
    for m in range(N + 1):
        c[m] = (2.0 / N) * np.sum(y * np.cos(m * theta))
    c[0] *= 0.5
    c[-1] *= 0.5
    return c


# ---------------------------------------------------------------------------
# Solver Newton-Raphson para o BVP y'' = exp(y), y(±1)=1
# ---------------------------------------------------------------------------
def solve_bvp_cheb_newton(N: int = 96, tol: float = 1e-12, maxit: int = 50):
    """
    Resolve o BVP via colocalização de Chebyshev (D,D^2) + Newton.

    Retorna
    -------
    x : nós de Chebyshev-Lobatto
    y : solução aproximada em x
    R : resíduo R = y'' - exp(y) em x (nas bordas setado para 0)
    Rp: derivada do resíduo R'  D*R
    info : dicionário com métricas de convergência
    """
    D, x = cheb(N)
    D2 = D @ D

    # Chute inicial: constante 1 (satisfaz Dirichlet)
    y = np.ones(N + 1)
    it = 0
    for it in range(1, maxit + 1):
        # Resíduo não linear F(y) = D2 y - exp(y)
        F = D2 @ y - np.exp(y)

        # Impõe Dirichlet fortemente nas linhas de fronteira
        F[0] = y[0] - 1.0
        F[-1] = y[-1] - 1.0
```

```
        # Jacobiano J = D2 - diag(exp(y)), com linhas de fronteira como identidade
        J = D2 - np.diag(np.exp(y))
        J[0, :] = 0.0
        J[0, 0] = 1.0
        J[-1, :] = 0.0
        J[-1, -1] = 1.0

        # Passo de Newton
        dy = np.linalg.solve(J, -F)
        y = y + dy

        if np.linalg.norm(dy, ord=np.inf) < tol:
            break

    # Avalia resíduo final e sua derivada
    R = D2 @ y - np.exp(y)
    R[0] = 0.0
    R[-1] = 0.0
    Rp = D @ R

    info = {
        "N": N,
        "iterations": it,
        "step_inf_norm": float(np.linalg.norm(dy, ord=np.inf)),
        "residual_inf_norm": float(np.linalg.norm(R[1:-1], ord=np.inf)),
        "residual_L2_norm": float(np.linalg.norm(R[1:-1]) / math.sqrt(N - 1)),
    }
    return x, y, R, Rp, info


# ---------------------------------------------------------------------
# Interpolação bariocêntrica (para estudo de refinamento)
# ---------------------------------------------------------------------
def barycentric_weights_cheb(N: int) -> np.ndarray:
    """
    Pesos bariocêntricos para nós de Chebyshev-Lobatto.
    """
    w = np.ones(N + 1)
    w[0] = 0.5
    w[-1] = 0.5
    w = w * ((-1.0) ** np.arange(N + 1))
    return w


def barycentric_interpolate(xnodes, w, fvals, xeval):
    """
    Interpolação de Lagrange na forma bariocêntrica (avaliando em xeval).
    """
    xnodes = np.asarray(xnodes)
    w = np.asarray(w)
    fvals = np.asarray(fvals)
    xeval = np.asarray(xeval)
    out = np.empty_like(xeval, dtype=float)
```

```
    for i, xv in enumerate(xeval):
        diff = xv - xnodes
        j = np.where(np.abs(diff) < 1e-14)[0]
        if j.size > 0:
            out[i] = fvals[j[0]]
        else:
            tmp = w / diff
            out[i] = np.sum(tmp * fvals) / np.sum(tmp)
    return out


# ---------------------------------------------------------------------
# Rotina principal (gera saídas de tabelas e figuras)
# ---------------------------------------------------------------------
def main():
    # Pastas de saída relativas a este arquivo
    here = Path(__file__).resolve().parent
    figdir = (here.parent / "figures")
    tabledir = (here.parent / "tables")
    figdir.mkdir(parents=True, exist_ok=True)
    tabledir.mkdir(parents=True, exist_ok=True)

    # 1) Resolver com N=96
    x, y, R, Rp, info = solve_bvp_cheb_newton(N=96)

    # 2) Salvar tabelas
    df = pd.DataFrame({"x": x, "y": y, "R": R, "Rprime": Rp})
    df.to_csv(tabledir / "ex1_solution_samples.csv", index=False)
    with open(tabledir / "ex1_summary.json", "w") as f:
        json.dump(info, f, indent=2)

    # 3) Figuras
    plt.figure()
    plt.plot(x, y)
    plt.scatter(x, y, s=12)
    plt.xlabel("x"); plt.ylabel("y(x)"); plt.title("Exercício 1: y(x) com pontos")
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.savefig(figdir / "ex1_solution_with_points.pdf", bbox_inches="tight")
    plt.close()

    plt.figure()
    plt.scatter(x, y, s=16)
    plt.xlabel("x"); plt.ylabel("y nos nós");
    ↪  plt.title("Exercício 1: nós de Chebyshev")
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.savefig(figdir / "ex1_points_only.pdf", bbox_inches="tight")
    plt.close()

    plt.figure()
    plt.plot(x, R)
    plt.xlabel("x"); plt.ylabel("R(x)"); plt.title("Exercício 1: Resíduo")
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.savefig(figdir / "ex1_residual.pdf", bbox_inches="tight")
```

```
    plt.close()

    plt.figure()
    plt.plot(x, Rp)
    plt.xlabel("x"); plt.ylabel("R'(x)");
    ↪  plt.title("Exercício 1: Derivada do Resíduo")
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.savefig(figdir / "ex1_residual_prime.pdf", bbox_inches="tight")
    plt.close()

    c = cheb_coeffs(y)
    plt.figure()
    plt.semilogy(np.arange(len(c)), np.abs(c))
    plt.xlabel("índice k"); plt.ylabel("|c_k|");
    ↪  plt.title("Exercício 1: decaimento espectral")
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.savefig(figdir / "ex1_coeff_decay.pdf", bbox_inches="tight")
    plt.close()

    # 4) Estudo de refinamento simples (N1=48 vs N2=96) em malha fina M=200
    N1, N2 = 48, 96
    x1, y1, *_ = solve_bvp_cheb_newton(N=N1)
    w1 = barycentric_weights_cheb(N1)
    w2 = barycentric_weights_cheb(N2)
    xfine = np.cos(np.pi * np.arange(201) / 200)

    y1f = barycentric_interpolate(x1, w1, y1, xfine)
    y2f = barycentric_interpolate(x, w2, y, xfine)
    diff = np.abs(y2f - y1f)
    df_ref = pd.DataFrame([{
        "N1": N1, "N2": N2,
        "Linf_on_fine": float(np.max(diff)),
        "L2_on_fine": float(np.linalg.norm(diff) / math.sqrt(len(diff)))
    }])
    df_ref.to_csv(tabledir / "ex1_refinement_study.csv", index=False)

    print("Concluído. Saídas gravadas em:")
    print(" -", figdir)
    print(" -", tabledir)


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

A.2 Exercício 2 — Sistema Não Linear: Newton–Jacobian e `fsolve`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Exercício 2 (PTC5725) - Sistema não linear:
    f1(x,y) = x^3 + y - 1 = 0
```

```
    f2(x,y) = y^3 - x + 1 = 0
Métodos: Newton-Jacobian (com e sem amortecimento) e fsolve (SciPy).
"""
import numpy as np, matplotlib.pyplot as plt, pandas as pd, importlib.util
from pathlib import Path

def F(v):
    x, y = v
    return np.array([x**3 + y - 1.0, y**3 - x + 1.0])

def J(v):
    x, y = v
    return np.array([[3*x**2, 1.0], [-1.0, 3*y**2]])

def newton_jacobian(x0, tol=1e-12, maxit=50, damping=False):
    x = np.array(x0, dtype=float)
    for it in range(maxit):
        f = F(x); normF = np.linalg.norm(f)
        if normF < tol:
            return x, True, it
        dx = np.linalg.solve(J(x), -f)
        if damping:
            alpha = 1.0
            while alpha > 1e-6:
                if np.linalg.norm(F(x + alpha*dx)) < normF:
                    x = x + alpha*dx; break
                alpha *= 0.5
            else:
                x = x + dx
        else:
            x = x + dx
    return x, False, maxit

def main():
    root = Path(__file__).resolve().parents[1]
    figdir, tabledir = root/'figures', root/'tables'
    figdir.mkdir(exist_ok=True, parents=True); tabledir.mkdir(exist_ok=True,
    ↪  parents=True)
    inits = [(0.5,0.5),(1.5,0.5),(-0.5,0.5),(2.0,-1.0),(-2.0,2.0)]
    rows = []
    for x0 in inits:
        sol, ok, it = newton_jacobian(x0, damping=False)
        rows.append({'method':'Newton','x0':x0[0],'y0':x0[1],'x':sol[0],'y':sol[1],
        ↪  'ok':ok,'it':it})
        sol, ok, it = newton_jacobian(x0, damping=True)
        rows.append({'method':'Newton (damped)','x0':x0[0],'y0':x0[1],'x':sol[0],
        ↪  'y':sol[1],'ok':ok,'it':it})
    if importlib.util.find_spec('scipy'):
        from scipy.optimize import fsolve
        sol,info,ier,msg = fsolve(F,[0.5,0.5],fprime=J,xtol=1e-12,full_output=True)
        rows.append({'method':'fsolve','x0':0.5,'y0':0.5,'x':sol[0],'y':sol[1],'ok'
        ↪  :ier==1,'it':info.get('nfev',0)})
```

```
    df = pd.DataFrame(rows);
    ↪  df.to_csv(tabledir/'ex2_newton_vs_fsolve_results.csv',index=False)
    xx=np.linspace(-3,3,300);yy=np.linspace(-3,3,300);X,Y=np.meshgrid(xx,yy)
    F1=X**3+Y-1;F2=Y**3-X+1
    plt.contour(X,Y,F1,[0]);plt.contour(X,Y,F2,[0],linestyles='--')
    for x0 in inits:
        s,_,_=newton_jacobian(x0,True);plt.plot(s[0],s[1],'o')
    plt.plot([1],[0],'r*',ms=10);plt.savefig(figdir/'ex2_contours.pdf');plt.close()
if __name__=='__main__': main()
```

A.3 Exercício 3 — Sistema 3D: Newton–Jacobian e `fsolve`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Exercício 3 (PTC5725) - Sistema 3D:
    f1(x,y,z) = sin(xy) + exp(-xz) - 0.9 = 0
    f2(x,y,z) = z*sqrt(x^2 + y^2) - 6.7 = 0
    f3(x,y,z) = tan(y/x) + cos(z) + 3.2 = 0
⌋
↪  Solução por Newton-Jacobian 3D (com e sem amortecimento) e alternativa com fsolve (SciPy, se dis
⌋
↪  Saídas: figures/ex3_convergence_normF.pdf, figures/ex3_state_trajectory.pdf; tables/ex3_3d_newto
"""

import numpy as np, pandas as pd, matplotlib.pyplot as plt, importlib.util
from pathlib import Path

def F(v):
    x, y, z = v
    return np.array([
        np.sin(x*y) + np.exp(-x*z) - 0.9,
        z*np.sqrt(x**2 + y**2) - 6.7,
        np.tan(y/x) + np.cos(z) + 3.2
    ], dtype=float)

def J(v):
    x, y, z = v
    r = np.sqrt(x**2 + y**2)
    f1x = np.cos(x*y)*y - z*np.exp(-x*z)
    f1y = np.cos(x*y)*x
    f1z = -x*np.exp(-x*z)
    df2dx = 0.0 if r==0 else z*(x/r)
    df2dy = 0.0 if r==0 else z*(y/r)
    df2dz = r
    sec2 = 1.0/np.cos(y/x)**2 if x!=0 else np.inf
    f3x = sec2 * (-y/(x**2)) if x!=0 else np.inf
    f3y = sec2 * (1.0/x) if x!=0 else np.inf
    f3z = -np.sin(z)
    return np.array([[f1x, f1y, f1z],
                     [df2dx, df2dy, df2dz],
```

```
                    [f3x, f3y, f3z]], dtype=float)

def newton3d(x0, tol=1e-10, maxit=100, damping=True):
    x = np.array(x0, dtype=float)
    hist = []
    for it in range(1, maxit+1):
        f = F(x); nF = float(np.linalg.norm(f, ord=2))
        hist.append({"it": it, "x": x.copy(), "normF": nF})
        if nF < tol:
            return x, True, it, hist
        try:
            dx = np.linalg.solve(J(x), -f)
        except np.linalg.LinAlgError:
            return x, False, it-1, hist
        if damping:
            alpha = 1.0; fx = nF
            for _ in range(40):
                xt = x + alpha*dx
                if np.linalg.norm(F(xt)) < fx:
                    x = xt; break
                alpha *= 0.5
            else:
                x = x + dx
        else:
            x = x + dx
    return x, False, maxit, hist

def main():
    root = Path(__file__).resolve().parents[1]
    figdir, tabledir = root/'figures', root/'tables'
    figdir.mkdir(parents=True, exist_ok=True); tabledir.mkdir(parents=True,
    ↪  exist_ok=True)

    # Rodar alguns chutes
    inits = [(1.0,2.0,2.0), (0.8,1.8,2.0), (1.2,2.2,2.0), (1.0,2.0,2.2),
    ↪  (1.0,2.0,1.8)]
    rows = []
    for x0 in inits:
        sol, ok, it, _ = newton3d(x0, damping=True)
        rows.append({"method":"Newton-3D (damped)","x0":x0[0],"y0":x0[1],"z0":x0[2],
                     "x":float(sol[0]),"y":float(sol[1]),"z":float(sol[2]),
                     ↪  "converged":ok,"iterations":it})
        sol2, ok2, it2, _ = newton3d(x0, damping=False)
        rows.append({"method":"Newton-3D (pure)","x0":x0[0],"y0":x0[1],"z0":x0[2],
                     "x":float(sol2[0]),"y":float(sol2[1]),"z":float(sol2[2]),
                     ↪  "converged":ok2,"iterations":it2})

    # fsolve se disponível
    if importlib.util.find_spec("scipy") is not None:
        from scipy.optimize import fsolve
        sol, info, ier, msg = fsolve(lambda v: F(v), np.array([1.0,2.0,2.0]),
        ↪  fprime=lambda v: J(v), xtol=1e-12, full_output=True)
        rows.append({"method":"fsolve","x0":1.0,"y0":2.0,"z0":2.0,
```

```
                    "x":float(sol[0]),"y":float(sol[1]),"z":float(sol[2]),
                    "converged":bool(ier==1),"iterations":int(info.get("nfev",
                    0))})

    df = pd.DataFrame(rows)
    df.to_csv(tabledir/'ex3_3d_newton_vs_fsolve_results.csv', index=False)

    # Convergência
    _, _, _, hist = newton3d((1.0,2.0,2.0), damping=True)
    its = [h["it"] for h in hist]; norms = [h["normF"] for h in hist]
    plt.figure(); plt.semilogy(its, norms, '-o'); plt.xlabel('Iteração');
    plt.ylabel('||F||_2'); plt.grid(True, linestyle='--', alpha=0.5)
    plt.title('Convergência do Newton 3D (amortecido)');
    plt.savefig(figdir/'ex3_convergence_normF.pdf', bbox_inches='tight');
    plt.close()

    xs = [h["x"][0] for h in hist]; ys = [h["x"][1] for h in hist]; zs = [h["x"][2]
    for h in hist]
    plt.figure(); plt.plot(its, xs, '-o', label='x_k'); plt.plot(its, ys, '-o',
    label='y_k'); plt.plot(its, zs, '-o', label='z_k')
    plt.xlabel('Iteração'); plt.ylabel('Valor'); plt.grid(True, linestyle='--',
    alpha=0.5); plt.legend()
    plt.title('Trajetória das variáveis (Newton 3D amortecido)');
    plt.savefig(figdir/'ex3_state_trajectory.pdf', bbox_inches='tight');
    plt.close()

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

B Setup ambiente Python

Versão da linguagem. Os experimentos foram desenvolvidos e testados em:

- **Python:** 3.12.10 (64 bits)
- **Sistema operacional:** Linux (WSL2) / compatível com Windows, macOS e distribuições baseadas em Debian.

Dependências principais. A seguir listam-se os pacotes necessários e suas respectivas versões utilizadas durante os experimentos:

Tabela 5: Pacotes e versões utilizados nos exercícios.

Biblioteca	**Versão recomendada**
`numpy`	1.26.0
`scipy`	1.14.1
`pandas`	2.2.2
`matplotlib`	3.9.2
`minted` (para LaTeX)	2.9
`Pygments` (para sintaxe colorida no LaTeX)	2.18.0
`python3-tk` (opcional, para gráficos locais)	–

Criação do ambiente virtual. Para garantir a reprodutibilidade, recomenda-se criar um ambiente virtual isolado:

```
# Criar o ambiente virtual
python3 -m venv venv

# Ativar o ambiente (Linux/Mac)
source venv/bin/activate

# Ativar o ambiente (Windows)
venv\Scripts\activate
```

Instalação das dependências. Após ativar o ambiente, instalar os pacotes requeridos:

```
pip install numpy==1.26.0 scipy==1.14.1 pandas==2.2.2 matplotlib==3.9.2
```

Estrutura esperada do projeto. Após a configuração, a estrutura de diretórios deve ser:

```
PTC5725_Tarefa04/
 code/
    ex1_cheb_newton.py
    ex2_newton_vs_fsolve.py
    ex3_newton3d_vs_fsolve.py
 figures/
 tables/
 refs/
    refs.bib
 main.tex
```

Execução dos scripts. Cada script Python pode ser executado individualmente para gerar suas figuras e tabelas:

```
# Executar o exercício 1
python code/ex1_cheb_newton.py

# Executar o exercício 2
python code/ex2_newton_vs_fsolve.py

# Executar o exercício 3
python code/ex3_newton3d_vs_fsolve.py
```

As saídas (figuras e tabelas) são automaticamente salvas nas pastas correspondentes. Após gerar todas as saídas, o relatório pode ser compilado com LaTeX:

```
pdflatex main.tex
bibtex main
pdflatex main.tex
pdflatex main.tex
```

Referências

[1] Osvaldo Guimarães. Aula 04 – introdução aos métodos espectrais, 2025. Slides fornecidos pelo professor da disciplina PTC5725.

[2] Ángel Franco García. Sistema de ecuaciones no lineales. `https://www.sc.ehu.es/sbweb/fisica3/numerico/raices/raices_5.html`, 2016. Material de apoio utilizado na disciplina.