# Atividade 1 (MAP0214) - 20 de agosto de 2025

Vitor Nirschl | vitor.nirschl@usp.br | Nº USP 13685771

O modelo a ser utilizado nessa atividade é a Lei de resfriamento de Newton, que trata da perda de temperatura $T(t)$ de um corpo, com capacidade térmica constante, para um ambiente com temperatura constante $T_{\text{amb}}$. Ele é dado pela equação diferencial ordinária de primeira ordem

$$\frac{\mathrm{d}T}{\mathrm{d}t} = r(T(t) - T_{\text{amb}})$$

em que $r$ é o coeficiente de perda térmica, dado em unidades inversas de tempo.

Estamos interessados em fazer uma aproximação numérica para esse problema. Se reescrevemos

$$\frac{\mathrm{d}T}{\mathrm{d}t} = f(t, T(t))$$

então temos

$$f(t, T(t)) = r(T(t) - T_{\text{amb}})$$

Sendo $t_0$ um instante inicial em que conheçemos $T(t_0) \equiv T_0$, consideramos o intervalo de tempo de interesse $\left[t_0, t_f\right]$ e o particionamos em $n$ intervalos $(t_k, t_{k+1})$, em que

$$t_{k+1} = t_k + \Delta t \text{ com } k = 0, 1, ...n-1, \text{sendo } \Delta t = \frac{t_f - t_0}{n}$$

Sendo $T_k$ a aproximação numérica de $T(t_k)$, teremos

$$T_{k+1} = T_k + \Delta t \phi_{\text{RK4}}(t_k, T_k, \Delta t)$$

em que $\phi_{\text{RK4}}$ é a função de discretização dada pelo método de Runge-Kutta clássico

$$\phi_{\text{RK4}}(t_k, T_k, \Delta t) = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

em que

$$k_1(t_k, T_k, \Delta t) = f(t_k, T_k)$$

$$k_2(t_k, T_k, \Delta t) = f\left(t_k + \frac{\Delta t}{2}, T_k + \frac{\Delta t}{2} k_1\right)$$

$$k_3(t_k, T_k, \Delta t) = f\left(t_k + \frac{\Delta t}{2}, T_k + \frac{\Delta t}{2} k_2\right)$$

$$k_4(t_k, T_k, \Delta t) = f(t_k + \Delta t, T_k + \Delta t k_3)$$

Nossa EDO tem solução exata, que poderemos utilizar para avaliar a qualidade da aproximação feita. Ela é dada por

$$T(t) = T_0 e^{rt} + T_{\text{amb}}(1 - e^{rt})$$

Implementando em Python o código exibido no Apêndice, utilizamos como parâmetros $r = -0.5\text{s}^{-1}$, $T(0) \equiv T_0 = 373\text{K}$ e $T_{\text{amb}} = 273\text{K}$, e o processo foi estudado num intervalo de 0s a 10s. O passo de integração escolhido foi $\Delta t = 0.1\text{s}$, observando graficamente a convergência das aproximações por passos de integração sucessivamente, em que os pontos cada vez mais sobrepuseram-se. Além disso, foi feita a comparação da aproximação com a solução exata.

A Tabela 1 apresenta 20 pontos das temperaturas $T_K$ do corpo aproximadas em cada instante $t_k$.

| Tempo (s) | Temperatura (K) |
|---|---|
| 0.50 | 350.88 |
| 0.60 | 347.08 |
| 0.70 | 343.47 |
| 0.80 | 340.03 |
| 0.90 | 336.76 |
| 1.00 | 333.65 |
| 1.10 | 330.69 |
| 1.20 | 327.88 |
| 1.30 | 325.20 |
| 1.40 | 322.66 |
| 1.50 | 320.24 |
| 1.60 | 317.93 |
| 1.70 | 315.74 |
| 1.80 | 313.66 |
| 1.90 | 311.67 |
| 2.00 | 309.79 |
| 2.10 | 307.99 |
| 2.20 | 306.29 |
| 2.30 | 304.66 |
| 2.40 | 303.12 |

Tabela 1: Resultados de 20 dos pontos obtidos com a aproximação de Runge-Kutta clássica para o resultado da lei de resfriamento de Newton, utilizando passo de integração de $\Delta t = 0.1s$.

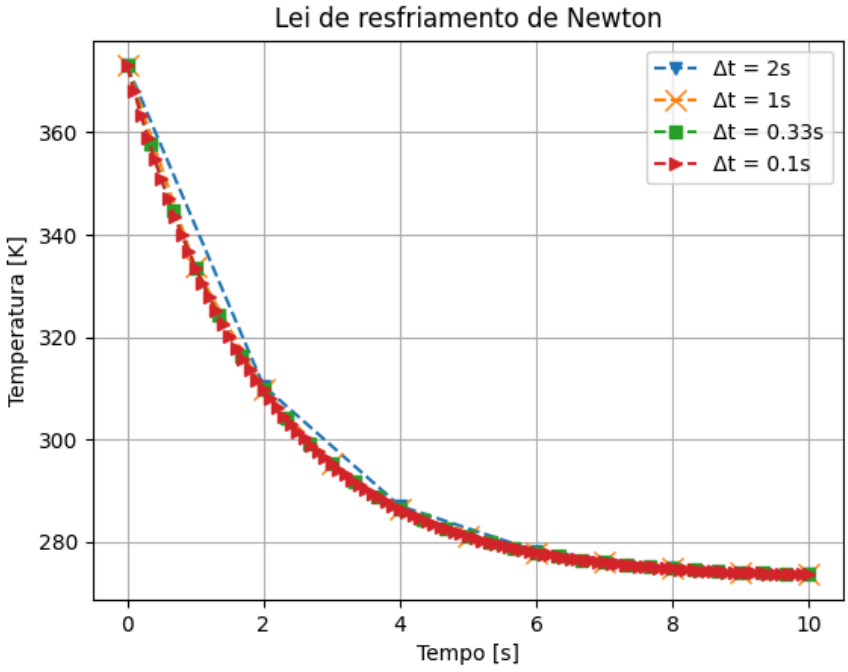


Figura 1: Aproximações com passos de integração sucessivamente menores para o problema, sendo aquela com $\Delta t = 0.1s$ a escolhida.

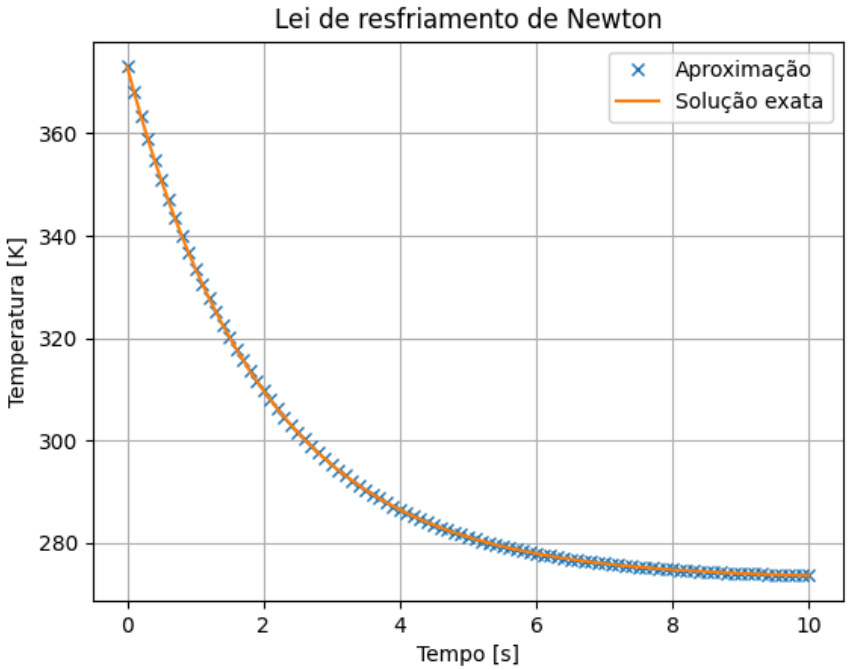


Figura 2: Comparação da melhor aproximação, com passo de integração $\Delta t = 0.1s$, com a solução exata do problema (linha contínua).

## Apêndice: Código utilizado

Todo o código aqui apresentado e utilizado durante a atividade foi integralmente desenvolvido por mim, **sem** utilização de auxilio de inteligência artifícial generativa ou de outras espécies.

```py
'''
Método de um passo explícito para a atividade 1 de cálculo numérico
'''
# imports

import numpy as np

# funções de consistência

def verifica_tempo(tempo_inicial, tempo_final):
    '''
    Verifica que os tempos fornecidos são números; garante a ordenação temporal.
    '''
    if tempo_inicial >= tempo_final:
        raise ValueError("O tempo inicial deve ser menor do que o tempo final.")
    if (not isinstance(tempo_inicial, (int, float))) or (not isinstance(tempo_final, (int, float))):
        raise TypeError("Os extremos do intervalo devem ser números.")

def verifica_temperaturas(temperatura_corpo, temperatura_ambiente):
```

```
20      '''
21      Verifica que as temperaturas são números e garante que a
22      temperatura ambiente seja inferior à do corpo.
23      '''
24      if (not isinstance(temperatura_ambiente, (int, float))) or (not
25          isinstance(temperatura_corpo, (int, float))):
26          raise TypeError("As temperaturas devem ser números.")
27      if not temperatura_corpo > temperatura_ambiente:
28          raise ValueError("A temperatura do corpo deve ser maior do que a
            temperatura ambiente.")
29
30  def verifica_coeficiente(r):
31      '''
32      Verifica que o coeficiente de perda térmica é um
33      número negativo.
34      '''
35      message = "O coeficiente de perda térmica deve ser um número negativo."
36      if not isinstance(r, (int, float)):
37          raise TypeError(message)
38      if not r < 0:
39          raise ValueError(message)
40
41  def verifica_passo_integracao(dt):
42      '''
43      Verifica que o passo de integração é um número positivo.
44      '''
45      message = "O passo de inegração deve ser um número positivo."
46      if not isinstance(dt, (int, float)):
47          raise TypeError(message)
48      if not dt > 0:
49          raise ValueError(message)
50
51  # funções do programa
52
53  def partition_interval(t_start, t_end, n):
54      '''
55      Particiona um intervalo [t_start, t_end] em n intervalos.
56      '''
57      # consistências
58      verifica_tempo(t_start, t_end)
59      if not isinstance(n, int) or n <= 0:
60          raise TypeError("O número de passos deve ser um número inteiro
            positivo.")
61
62      # time-steps
63      dt = (t_end - t_start)/n
```

```
64
65      # pontos de partição
66      partitioned_interval = np.linspace(t_start, t_end, n+1)
67      return dt, partitioned_interval
68
69  def f(r, temperature, amb_temperature):
70      '''
71      Derivada da temperatura temperature(t) num instante arbitrário t.
72      '''
73      # consistências
74      verifica_coeficiente(r)
75      verifica_temperaturas(temperature, amb_temperature)
76      return r * (temperature - amb_temperature)
77
78  def rk4(r, dt, temperature_k, amb_temperature):
79      '''
80      Função de discretização runge-kutta clássica (RK4) para a lei de
        resfriamento de Newton.
81      - r é o coeficiente de perda térmica
82      - dt é o passo de integração
83      - temperature_k é a aproximação da temperatura do corpo no instante t_k
84      - amb_temperature é a temperatura ambiente
85      '''
86      # consistências
87      verifica_passo_integracao(dt)
88      verifica_coeficiente(r)
89      verifica_temperaturas(temperature_k, amb_temperature)
90
91      # parâmetros auxiliares
92      k1 = f(r, temperature_k, amb_temperature)
93      k2 = f(r, temperature_k + (dt * k1)/2, amb_temperature)
94      k3 = f(r, temperature_k + (dt * k2)/2, amb_temperature)
95      k4 = f(r, temperature_k + dt * k3, amb_temperature)
96
97      return (k1 + 2 * k2 + 2 * k3 + k4)/6
98
99  def explicit_one_step(r, start_temperature, amb_temperature, dt,
    partitioned_interval):
100     '''
101     Algoritmo de um passo explícito para a Lei de Resfriamento de Newton.
102     r é o coeficiente de perda térmica
103     start_temperature é a temperatura inicial do corpo
104     amb_temperature é a temperatura ambiente
105     dt é o passo de integração
106     partitioned_interval é o intervalo de tempo particionado
107     '''
```

```
    # consistências
    verifica_coeficiente(r)
    verifica_passo_integracao(dt)
    verifica_temperaturas(start_temperature, amb_temperature)
    if not isinstance(partitioned_interval, np.ndarray):
        raise TypeError("partitioned_interval deve ser um ndarray do numpy.")

    t0 = partitioned_interval[0]
    approx_table = np.array([[t0, start_temperature]])
    temperature = start_temperature
    for i in range(1, partitioned_interval.size):
        if temperature > amb_temperature:
            temperature = temperature + dt * rk4(r, dt, temperature,
            amb_temperature)
            approx_table = np.append(approx_table, [[partitioned_interval[i],
            temperature]], 0)
        else: break

    return approx_table


if __name__ == "__main__":
    # Intervalo de estudo
    TIME_START = 0
    TIME_END = 2

    # Quantidade de partições
    N = 50

    # Temperatura inicial do corpo e temperatura ambiente
    INITIAL_TEMP = 100
    AMBIENT_TEMP = 10

    # Coeficiente de perda térmica
    R = -1

    delta_t, interval_partitioned = partition_interval(TIME_START, TIME_END, N)
    table = explicit_one_step(R, INITIAL_TEMP, AMBIENT_TEMP, delta_t,
    interval_partitioned)

    print(table)
    print(table.transpose())
```

py
```
'''
Gera tabelas e gráficos para a atividade 1 de cálculo numérico
'''

```

```python
# imports

import numpy as np
import explicit_one_step as eos
import matplotlib.pyplot as plt

# funções do programa

def gera_tabela_typst():
    '''
    Recebe array (n, 1) do numpy, com n inteiro positivo, e retorna
    tabela formatada do typst, com cabeçalho (t, T(t)).
    '''
    return

def funcao_exata(params, t):
    '''
    Solução exata para a Lei de Resfriamento de Newton, em que
    params[0] é a temperatura ambiente,
    params[1] é a temperatura inicial do corpo,
    params[2] é o coeficiente de perda térmica,
    t é uma array do numpy com o intervalo de tempo discretizado.
    '''
    # consistências
    ambient_temperature = params[0]
    start_temperature = params[1]
    r = params[2]

    eos.verifica_temperaturas(start_temperature, ambient_temperature)
    eos.verifica_coeficiente(r)

    return start_temperature * np.exp(r * t) + ambient_temperature * (1 - np.exp(r * t))

if __name__ == "__main__":
    STEPS_NUMBERS = [5, 10, 15, 30, 100]
    STEPS_SIZES = []
    TABLES = []

    R = -.5
    START_TEMPERATURE = 373
    AMBIENT_TEMPERATURE = 273

    TIME_START = 0
    TIME_END = 10

```

```
50      # aproximações
51      for number in STEPS_NUMBERS:
52          dt, intervals = eos.partition_interval(TIME_START, TIME_END, number)
53          STEPS_SIZES.append(dt)
54          APPROX_TABLE = eos.explicit_one_step(R, START_TEMPERATURE,
            AMBIENT_TEMPERATURE, dt,
55              intervals)
56          TABLES.append(APPROX_TABLE)
57
58      # função exata
59      TIME_ARRAY = np.linspace(TIME_START, TIME_END, 500)
60      Y = funcao_exata([AMBIENT_TEMPERATURE, START_TEMPERATURE, R], TIME_ARRAY)
61
62      # gráfico de aproximações sucessivas
63      fig1, ax1 = plt.subplots()
64
65      ax1.plot(TABLES[0].transpose()[0], TABLES[0].transpose()[1], label="Δt =
        2s", marker="v", linestyle="--")
66      ax1.plot(TABLES[1].transpose()[0], TABLES[1].transpose()[1], label="Δt =
        1s", marker="x", linestyle="--", markersize="10")
67      ax1.plot(TABLES[3].transpose()[0], TABLES[3].transpose()[1], label="Δt =
        0.33s", marker="s", linestyle="--")
68      ax1.plot(TABLES[4].transpose()[0], TABLES[4].transpose()[1], label="Δt =
        0.1s", marker=">", linestyle="--")
69
70      ax1.set_xlabel('Tempo [s]')
71      ax1.set_ylabel('Temperatura [K]')
72      ax1.set_title('Lei de resfriamento de Newton')
73      ax1.legend()
74      ax1.grid()
75      fig1.savefig('aproximacoes.png')
76
77      # gráfico da aproximação final com resultado exato
78      fig2, ax2 = plt.subplots()
79
80      ax2.plot(TABLES[4].transpose()[0], TABLES[4].transpose()[1],
        label="Aproximação", marker="x", linestyle="None")
81      ax2.plot(TIME_ARRAY, Y, label="Solução exata")
82      ax2.set_xlabel('Tempo [s]')
83      ax2.set_ylabel('Temperatura [K]')
84      ax2.set_title('Lei de resfriamento de Newton')
85      ax2.legend()
86      ax2.grid()
87      fig2.savefig('comparacao_com_exata.png')
88
89      # tabela com todos os valores da aproximação final
90      data = ""
```

```
    for point in TABLES[4]:
        data += f"[{point[0]:.2f}], [{point[1]:.2f}],\n"

    with open("tabela.txt", "w") as f:
        f.write(data)
```