MAP214 -Cálculo Numérico com Aplicações em Física
EP3

Guilherme Pacheco Paredes N°USP: 12693754

Instituto de Física, Universidade de São Paulo - SP

31/10/2024

Os exercícios computacionais deste EP foram feitos utilizando a linguagem de prgramação *Python*. As bibliotecas *Numpy* e *matplotlib.pyplot* foram utilizadas para operações matemáticas e plots dos gráficos.

1

Para resolver a integral de forma numérica iremos construir um programa em precisão simples utilizando o comando do *Python float32*.

```
import numpy as np
import matplotlib.pyplot as plt

# Função a ser integrada
def f(x):
    return np.float32(6 - 6 * x ** 5)

I_analitico = np.float32(5)

# Função para o método de Simpson
def simpson_integral(a, b, N):
    # Passo de integração em precisão simples
    h = np.float32((b - a) / N)
    x_values = [np.float32(a + i * h) for i in range(N + 1)]
    y_values = [f(x) for x in x_values]

    # Cálculo da integral usando método de Simpson
    integral = np.float32(0)
    for i in range(1, N, 2):  # Termos ímpares
        integral += np.float32(4) * y_values[i]
    for i in range(2, N - 1, 2):  # Termos pares
        integral += np.float32(2) * y_values[i]

    # Acrescenta os extremos
    integral += y_values[0] + y_values[N]
    integral *= h / np.float32(3)
    return integral

# Parâmetros
a, b = np.float32(0), np.float32(1)
p_values = np.arange(1, 26, dtype=int)

# Tabela de resultados e cálculo dos erros
print("-" * 55)
print(f"|{'p':^10}|{'N':^10}|{'I_num':^15}|{'Erro':^15}|")
print("-" * 55)

# Cálculo da integral e análise de erro ponto a ponto
erros = []
for p in p_values:
    N = 2 ** p
```

```
    I_num = simpson_integral(a, b, N)

    erro = abs(I_num - I_analitico)

    erros.append((p, N, I_num, erro))
    print(f"|{p:^10}|{N:^10}|{I_num:^15.8f}|{erro:^15.8f}|")

print("-" * 55)
```

a) Com $N = 2^p$ sendo o número de intervalos e com erro $= |I_{num} - I|$. A saída com a tabela:

```
-------------------------------------------------------
|    p     |    N     |     I_num     |     Erro      |
-------------------------------------------------------
|    1     |    2     |  4.87500000   |  0.12500000   |
|    2     |    4     |  4.99218750   |  0.00781250   |
|    3     |    8     |  4.99951172   |  0.00048828   |
|    4     |    16    |  4.99996948   |  0.00003052   |
|    5     |    32    |  4.99999809   |  0.00000191   |
|    6     |    64    |  5.00000048   |  0.00000048   |
|    7     |   128    |  5.00000000   |  0.00000000   |
|    8     |   256    |  4.99999905   |  0.00000095   |
|    9     |   512    |  5.00000238   |  0.00000238   |
|    10    |   1024   |  5.00000143   |  0.00000143   |
|    11    |   2048   |  5.00000191   |  0.00000191   |
|    12    |   4096   |  5.00000572   |  0.00000572   |
|    13    |   8192   |  5.00001240   |  0.00001240   |
|    14    |  16384   |  5.00003529   |  0.00003529   |
|    15    |  32768   |  5.00007439   |  0.00007439   |
|    16    |  65536   |  5.00019455   |  0.00019455   |
|    17    |  131072  |  5.00041580   |  0.00041580   |
|    18    |  262144  |  5.00104141   |  0.00104141   |
|    19    |  524288  |  5.00215244   |  0.00215244   |
|    20    | 1048576  |  5.00578117   |  0.00578117   |
|    21    | 2097152  |  5.01456404   |  0.01456404   |
|    22    | 4194304  |  5.03342247   |  0.03342247   |
|    23    | 8388608  |  4.60506296   |  0.39493704   |
|    24    | 16777216 |  5.33333349   |  0.33333349   |
|    25    | 33554432 |  5.30003234   |  0.30003214   |
-------------------------------------------------------
```

Foi obtido $I_{num} = 5.30003234$ para o método de Simpson na última iteração. é possível notar que ao aumentar o número de operações atinge-se um ponto onde os erros de arredondamento *Roundoff* começam a dominar, resultando em maiores erros acumulados.

b) Realizando agora o mesmo em precisão dupla, alterando de $np.float32$ para $np.float64$ (ou realizando no modo padrão do *Python* que já realiza as operações em 64 bits), temos a seguinte tabela como saída:

```
-------------------------------------------------------
|    p     |    N     |     I_num     |     Erro      |
-------------------------------------------------------
|    1     |    2     |  4.87500000   |  0.12500000   |
|    2     |    4     |  4.99218750   |  0.00781250   |
|    3     |    8     |  4.99951172   |  0.00048828   |
|    4     |    16    |  4.99996948   |  0.00003052   |
|    5     |    32    |  4.99999809   |  0.00000191   |
|    6     |    64    |  4.99999988   |  0.00000012   |
|    7     |   128    |  4.99999999   |  0.00000001   |
|    8     |   256    |  5.00000000   |  0.00000000   |
|    9     |   512    |  5.00000000   |  0.00000000   |
|    10    |   1024   |  5.00000000   |  0.00000000   |
```

```
|    11    |   2048   |  5.00000000   |  0.00000000   |
|    12    |   4096   |  5.00000000   |  0.00000000   |
|    13    |   8192   |  5.00000000   |  0.00000000   |
|    14    |  16384   |  5.00000000   |  0.00000000   |
|    15    |  32768   |  5.00000000   |  0.00000000   |
|    16    |  65536   |  5.00000000   |  0.00000000   |
|    17    |  131072  |  5.00000000   |  0.00000000   |
|    18    |  262144  |  5.00000000   |  0.00000000   |
|    19    |  524288  |  5.00000000   |  0.00000000   |
|    20    | 1048576  |  5.00000000   |  0.00000000   |
|    21    | 2097152  |  5.00000000   |  0.00000000   |
|    22    | 4194304  |  5.00000000   |  0.00000000   |
|    23    | 8388608  |  5.00000000   |  0.00000000   |
|    24    | 16777216 |  5.00000000   |  0.00000000   |
|    25    | 33554432 |  5.00000000   |  0.00000000   |
-------------------------------------------------------
```

O valor final para a integral em precisão dupla pelo método de Simpson é $I_{num} = 5.00000000$. Agora, para construir um gráfico de log_2(erro) em função de p, em que os pontos que retornam erro $= 0$ são eliminados, tanto para precisão simples quanto para precisão dupla, foi implementado o seguinte programa aos programas anteriores:

```python
# Dados para o gráfico
p_values_simple = [p for p, N, I_num, erro in erros_simple if erro > 0]
erro_values_simple = [np.log2(erro) for p, N, I_num, erro in erros_simple if erro > 0]

p_values_double = [p for p, N, I_num, erro in erros_double if erro > 0]
erro_values_double = [np.log2(erro) for p, N, I_num, erro in erros_double if erro > 0]

# Região de erros pelo método e de roundoff (Simples)
p_values_1_simple = p_values_simple[:5]
erro_values_1_simple = erro_values_simple[:5]
p_values_2_simple = p_values_simple[5:11]
erro_values_2_simple = erro_values_simple[5:11]

# Região de erros pelo método e de roundoff (Dupla)
p_values_1_double = p_values_double[:11]
erro_values_1_double = erro_values_double[:11]
p_values_2_double = p_values_double[12:18]
erro_values_2_double = erro_values_double[12:18]

# Ajuste linear para a primeira região (Simples)
alpha1_simple, logA1_simple = np.polyfit(p_values_1_simple, erro_values_1_simple, 1)

# Ajuste linear para a segunda região (Simples)
alpha2_simple, logA2_simple = np.polyfit(p_values_2_simple, erro_values_2_simple, 1)

# Ajuste linear para a primeira região (Dupla)
alpha1_double, logA1_double = np.polyfit(p_values_1_double, erro_values_1_double, 1)

# Ajuste linear para a segunda região (Dupla)
alpha2_double, logA2_double = np.polyfit(p_values_2_double, erro_values_2_double, 1)


plt.figure(figsize=(10, 6))

# Plotando os erros em precisão simples e dupla
plt.scatter(p_values_simple, erro_values_simple, label='Erro (Simples)', marker='o',
            color='black', s=20)
plt.scatter(p_values_double, erro_values_double, label='Erro (Dupla)', marker='s',
            edgecolor='red', facecolor='none', s=60)
plt.plot(p_values_2_simple, alpha2_simple * np.array(p_values_2_simple) + logA2_simple,
         color='blue', linestyle='--', label=f'Ajuste Linear,  = {-alpha2_simple:.4f}')
```

```
plt.plot(p_values_1_double, alpha1_double * np.array(p_values_1_double) + logA1_double,
         color='black', label=f'Ajuste Linear,  = {-alpha1_double:.4f}')
plt.plot(p_values_2_double, alpha2_double * np.array(p_values_2_double) + logA2_double,
         color='green', linestyle='--', label=f'Ajuste Linear,  = {-alpha2_double:.4f}')

# Configurações do gráfico
plt.xlabel('p')
plt.ylabel('$\log_{2}(\mathrm{erro})$')
plt.legend()
plt.show()
```

E a saída com o gráfico

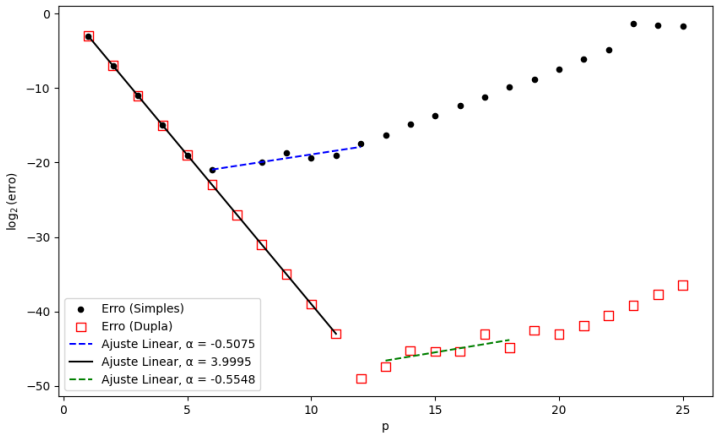

Figura 1: Erro do método de Simpson na integração da função $f(x) = 6 - 6x^5$, $0 < x < 1$. Foram obtidos os valores de coeficiente angular $\alpha = 3.9995$ para o erro do método e $\alpha = -0.5075$, $\alpha = -0.5548$ para os erros de *roundoff* em precisão simples e dupla, respectivamente.

O gráfico nos mostra que os erros são de fato $\mathcal{O}(h^\alpha)$ para o método utilizado, com o valor esperado $\alpha = 4$. Para o erro de *roundoff*, esperava-se $\mathcal{O}(\sqrt{N})$, em que $N = 1/h$ e $\alpha = -1/2$. De fato, foram obtidos valores muito próximos e precisos, como é mostrado no gráfico. É possível notar que para maiores valores de N a precisão diminui consideravelmente no erro de *roundoff* dado que eles são acumulados a cada operação.

2

O período de pêndulo simples para ângulos pequenos ($\theta_0 < 10\,\text{deg}$) é dado por $T_{Galileu} = 2\pi\sqrt{l/g}$. Quando esta aproximação não é mais válida, calculamos o período por

$$T = 4\sqrt{\frac{l}{g}}\int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2\sin^2\phi}}, \tag{2.1}$$

onde $k^2 \equiv [1 - \cos\theta_0]/2$, com θ_0 sendo o ângulo inicial. Com o intuito de obter a solução para esta integral de forma numérica por meio do método dos trapézios e então construir uma tabela com valores de θ_0, variando em $[0, \pi)$, e de $T/T_{Galileu}$, assim como um gráfico que relaciona esta razão em termos de θ_0, implementamos o seguinte código:

```python
import numpy as np
import matplotlib.pyplot as plt

# Constantes
l = 1.0
g = 9.81

# Função para calcular o valor da integral usando o método dos trapézios
def trapezoidal_integration(func, a, b, n, k):
    h = (b - a) / n
    integral = 0.5 * (func(a, k) + func(b, k))

    for i in range(1, n):
        integral += func(a + i * h, k)

    integral *= h
    return integral

# Função para a integral
def integrando(phi, k):
    term = 1 - k ** 2 * np.sin(phi) ** 2
    if np.isclose(term, 0):     # Evitar divisão por zero
        return 0
    return 1 / np.sqrt(term)

# Função principal para calcular T/TGalileu
def razao_periodos(num_theta_valores):
    theta_values = np.linspace(0, np.pi, num_theta_values)
    T_ratio = []

    for theta0 in theta_values:
        k_squared = (1 - np.cos(theta0)) / 2
        integral_value = trapezoidal_integration(integrando, 0, np.pi / 2, 1000, k_quadrado)
        T = 4 * np.sqrt(l / g) * integral_value
        T_Galileu = 2 * np.pi * np.sqrt(l / g)
        T_ratio.append(T / T_Galileu)

    return theta_values, T_ratio

# Parâmetros
num_theta_values = 200
theta_values, T_ratio = ratio_periods(num_theta_values)

# Tabela com 10 valores de theta0
print("-" * 37)
print(f"| {'theta_0 (rad)':^15} | {'T/T_Galileu':^15} |")
print("-" * 37)
# 10 valores igualmente espaçados utilizando fatiamento [inicio:fim:passo]
```

```
for theta, razao in zip(theta_values[::num_theta_values // 10],
                        T_ratio[::num_theta_values // 10]):
    print(f"| {theta:^15.8f} | {ratio:^15.8f} |")
print("-" * 37)

plt.figure(figsize=(8, 6))
plt.plot(theta_values, T_ratio, label='$T/T_{Galileu}$', color='blue')
plt.xlabel('$theta_0$ [rad]')
plt.ylabel('$T/T_{Galileu}$')
plt.axhline(1, color='black', linewidth=0.5, linestyle='--')
plt.grid()
plt.legend()
plt.show()
```

A saída com a tabela:

```
-------------------------------------
|  theta_0 [rad]  |   T/T_Galileu   |
-------------------------------------
|   0.00000000    |   1.00000000    |
|   0.31573796    |   1.00015278    |
|   0.63147591    |   1.00233651    |
|   0.94721387    |   1.01109029    |
|   1.26295182    |   1.03262985    |
|   1.57868978    |   1.07455253    |
|   1.89442773    |   1.14723852    |
|   2.21016569    |   1.26821755    |
|   2.52590364    |   1.47543834    |
|   2.84164160    |   1.88971292    |
-------------------------------------
```

E o gráfico:

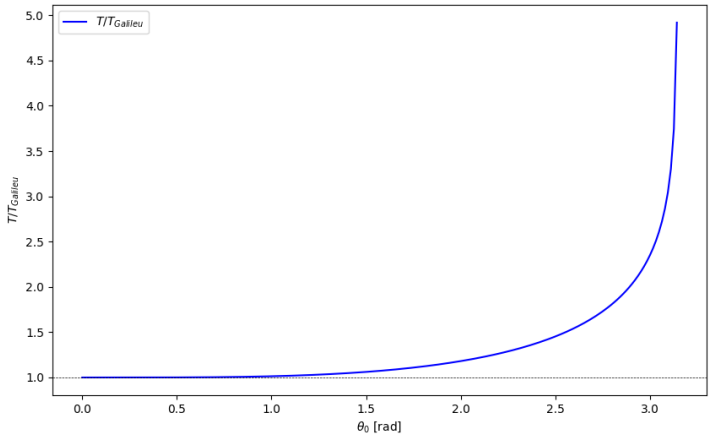

Figura 2: Relação $\theta_0 \times T/T_{Galileu}$. Em destaque uma linha tracejada na horizontal em $T/T_{Galileu}$, destacando para até quais valores de θ_0 os períodos são bem aproximados.

É possível ver que para valores próximos a $\theta_0 = 3$ rad a aproximação já se torna pouco eficaz, com $T_{Galileu}$ sendo quase metade de T para o último valor obtido na tabela.

3

Vamos obter a integral da função $y = 1 - x^2$, $0 < x < 1$ pelo método de Monte-Carlo.
a) Para criar uma rotina que retorne números aleatórios uniformemente distribuídos por *linear congruential generator*, o seguinte programa foi implementado:

```
# Parâmetros do gerador linear congruencial
a = 1103515245
c = 12345
m = 2147483647
seed = 12693754  # Semente inicial NUSP Z_0

def next_random(seed):
    new_seed = (a * seed + c) \% m     # Nova semente Z_(i+1)
    return new_seed / m, new_seed  # Valor normalizado U_i e nova semente Z_(i+1)
```

b) Façamos uma tentativa jogando 100 pontos (x, y), $0 < x < 1$ e $0 < y < 1$ aleatoriamente e determinar o valor da área sob a curva usando

$$I \approx \frac{\text{número de pontos dentro}}{\text{número total de pontos}}. \tag{3.1}$$

Para isso, vamos implementar o seguinte programa à rotina do item 3 a):

```
# Função da curva
def f(x):
    return 1 - x**2

# Função para estimar a área com 100 pontos usando o método Monte Carlo com o LCG
def estimate_area_lcg(n_points=100, seed=seed):
    count_inside = 0
    for _ in range(n_points):
        x_random, seed = next_random(seed)
        y_random, seed = next_random(seed)
        if y_random < f(x_random):
            count_inside += 1
    return count_inside / n_points, seed

# Parâmetros
n_points = 100

# Estimativa inicial da área
area_estimate, seed = estimate_area_lcg(n_points, seed)
print(f"Estimativa inicial da área sob a curva com 100 pontos: {area_estimate}")
```

E a saída:

```
Estimativa inicial da área sob a curva com 100 pontos: 0.65
```

Uma boa estimativa, dado $I_{analtico} = 2/3$.
c) Agora, dado N_t tentativas, em que cada tentativa joga 100 pontos aleatórios, vamos construir uma tabela contendo N_t, I_m (valor médio da integral), σ (desvio padrão) e σ_m (desvio padrão da média), tal que

$$I_m = \frac{1}{N_t} \sum_{i=1}^{N_t} I_i, \tag{3.2}$$

$$\sigma^2 = \frac{1}{N_t - 1} \sum_{i=1}^{N_t} (I_i - I_m)^2, \tag{3.3}$$

$\sigma_m = \sigma/\sqrt{N_t}$ e $I_m \pm \sigma_m$. Para isso, o seguinte código foi implementado utilizando novamente a rotina criada no item 3 a):

```
N_t_values = [2 ** i for i in range(1, 18)]  # 2, 4, 8, ..., 131072
results = []

# Cálculos para diferentes N_t
for N_t in N_t_values:
    estimates = []
    current_seed = seed
    for _ in range(N_t):
        area, current_seed = estimate_area_lcg(n_points, current_seed)
        estimates.append(area)

    # Cálculo de Im (média das integrais estimadas)
    I_m = sum(estimates) / N_t

    # Cálculo de ^2 (variância) manualmente
    variance = sum((I - I_m) ** 2 for I in estimates) / (N_t - 1)
    sigma = variance ** 0.5  # Desvio padrão ()

    # Cálculo de m (desvio padrão da média)
    sigma_m = sigma / (N_t ** 0.5)

    # Armazenar os resultados
    results.append((N_t, I_m, sigma, sigma_m))

# Exibir a tabela de resultados
print("-" * 50)
print(f"|{'N_t':^10}|{'I_m':^15}|{'sigma':^10}|{'sigma_m':^10}|")
print("-" * 50)
for N_t, I_m, sigma, sigma_m in results:
    print(f"|{N_t:^10}|{I_m:^15.6f}|{sigma:^10.6f}|{sigma_m:^10.6f}|")
print("-" * 50)
```

E a saída:

```
--------------------------------------------------
|    N_t    |      I_m      |   sigma   |   sigma_m  |
--------------------------------------------------
|     2     |   0.620000    | 0.014142  |  0.010000  |
|     4     |   0.647500    | 0.035000  |  0.017500  |
|     8     |   0.662500    | 0.043012  |  0.015207  |
|    16     |   0.675000    | 0.049126  |  0.012281  |
|    32     |   0.679375    | 0.048986  |  0.008660  |
|    64     |   0.673125    | 0.048136  |  0.006017  |
|   128     |   0.667813    | 0.049897  |  0.004410  |
|   256     |   0.665156    | 0.051136  |  0.003196  |
|   512     |   0.666270    | 0.050115  |  0.002215  |
|   1024    |   0.667051    | 0.048119  |  0.001504  |
|   2048    |   0.666904    | 0.046750  |  0.001033  |
|   4096    |   0.667434    | 0.046836  |  0.000732  |
|   8192    |   0.666992    | 0.047365  |  0.000523  |
|  16384    |   0.667032    | 0.047239  |  0.000369  |
|  32768    |   0.666982    | 0.047327  |  0.000261  |
|  65536    |   0.666691    | 0.047306  |  0.000185  |
|  131072   |   0.666553    | 0.047178  |  0.000130  |
--------------------------------------------------
```

Nota-se a boa precisão do método para obter numericamente a integral da função. Seja $I_{analtico} = 2/3 = 0.6666667$ com 7 dígitos, podemos ver que que a partir de $N_t = 32768$ o I_m mostra uma tendência a convergir para o valor analítico, bem como σ_m tendendo a ser cada vez menor.