MS211 - Cálculo Numérico

Projeto 2 - Equações Diferenciais Ordinárias

Letícia Lopes Mendes da Silva - RA: 184423
Iran Seixas Lopes Neto - RA: 244827

Parte I

Nos exercícios a seguir, vamos considerar os seguintes métodos para solucionar numericamente as equações diferenciais ordinárias (EDOs):

(a) Método de Runge-Kutta de 2ª Ordem (ou Método de Euler Aperfeiçoado):

$$y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2) \quad \begin{cases} k_1 = f(x_n, y_n), \\ k_2 = f\left(x_n + h, y_n + hk_1\right) \end{cases}$$

(b) Método de Runge-Kutta de 4ª Ordem:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad \begin{cases} k_1 = f(x_n, y_n), \\ k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \\ k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right), \\ k_4 = f\left(x_n + h, y_n + hk_3\right) \end{cases}$$

Item 1)

$$\begin{cases} y' = \dfrac{1}{x^2} - \dfrac{y}{x} - y^2 \\ y(1) = -1 \end{cases}$$

Dado o PVI acima, vamos formular códigos em `Python` dos métodos (a) e (b) para encontrar uma tabela com as soluções numéricas de y para $x \in [1, 2]$, com passo $h = 0.1$. As variáveis iniciais são:

```
d_y = lambda x, y: x**(-2) - y/x - y**2
x_0 = 1.0
x_f = 2.0
y_0 = -1.0
h = 0.1
```

(a) Método de Euler Aperfeiçoado:

```
def euler_aperfeicoado(d_y, x:float, x_f:float, y:float, h:float):
    f = [y]
    while x < x_f:
        k_1 = d_y(x, y)
        k_2 = d_y(x + h, y + h * k_1)

        y = y + (h/2) * (k_1 + k_2)
        f.append(y)

        x = x + h
    return f
```

```
f_a = euler_aperfeicoado(d_y, x_0, x_f, y_0, h)
for y in f_a:
    print(f'{y:.4f}')
```

i	x_i	y_i
0	1	-1.0000
1	1.1	-0.9083
2	1.2	-0.8319
3	1.3	-0.7672
4	1.4	-0.7118
5	1.5	-0.6638
6	1.6	-0.6218
7	1.7	-0.5847
8	1.8	-0.5517
9	1.9	-0.5222
10	2	-0.4956

Tabela 1: Solução numérica obtida pelo Método de Euler Aperfeiçoado.

(b) Método de Runge-Kutta de 4ª ordem:

```
def runge_kutta_4(d_y, x:float, x_f:float, y:float, h:float):
    f = [y]
    while x < x_f:
        k_1 = d_y(x, y)
        k_2 = d_y(x + h/2, y + (h/2) * k_1)
        k_3 = d_y(x + h/2, y + (h/2) * k_2)
        k_4 = d_y(x + h, y + h * k_3)

        y = y + (h/6) * (k_1 + 2*k_2 + 2*k_3 + k_4)
        f.append(y)

        x = x + h
    return f
```

```
f_b = runge_kutta_4(d_y, x_0, x_f, y_0, h)
for y in f_b:
    print(f'{y:.4f}')
```

i	x_i	y_i
0	1	-1.0000
1	1.1	-0.9091
2	1.2	-0.8333
3	1.3	-0.7692
4	1.4	-0.7143
5	1.5	-0.6667
6	1.6	-0.6250
7	1.7	-0.5882
8	1.8	-0.5556
9	1.9	-0.5263
10	2	-0.5000

Tabela 2: Solução numérica obtida pelo Método de Runge-Kutta de 4ª Ordem.

Item 2)

Apenas pelas tabelas, vemos que existem diferenças numéricas bem sutis em relação aos resultados dos dois métodos, nas casas centesimais. Vamos analisar graficamente essas soluções, comparando-as com a solução analítica para o problema: $y = -\frac{1}{x}$.

```
x = np.linspace(x_0, x_f, int((x_f - x_0) / h) + 1)
sol = -1/x

legend_a = "Euler Aperfeicoado"
legend_b = "Runge-Kutta de Ordem 4"
legend_sol = rf"Solucao Analitica: $y(x) = -\dfrac{{1}}{{x}}$"

plt.scatter(x, f_a, label=legend_a, color="tab:blue", s=100)
plt.scatter(x, f_b, label=legend_b, color="tab:red", s=100)
plt.plot(x, sol, label=legend_sol, color="tab:orange")
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

Analisando o gráfico 1, confirmamos que os pontos estão bem próximos entre si e se aproximam bem da curva da solução analítica. No entanto, ao analisar os valores com maior precisão, isto é, considerando as casas centesimais, o método de Runge-Kutta de Ordem 4 apresenta uma aproximação melhor que o de Euler Aperfeiçoado, sendo mais recomendado nestes casos.

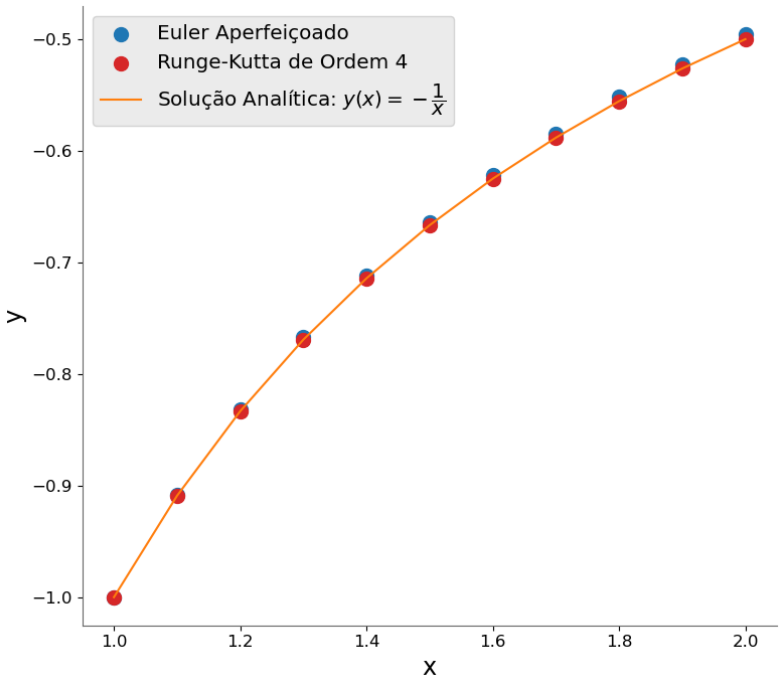

Figura 1: Soluções numéricas e analítica para o PVI dado.

Parte II

$$\begin{cases} P' = aP - bP^2 \\ P(0) = 76.1 \end{cases}$$

Para este exercício, criamos um código em `Python` que calcula a solução numérica do PVI acima usando a função `odeint` e a solução "exata" $P(t) = \dfrac{89.7617 \cdot e^{0.02t}}{1 + 0.1795 \cdot e^{0.02t}}$, para $t \in [0, 80]$ e passo $h = 10$, e criamos uma tabela com esses valores:

```
from scipy.integrate import odeint

d_P = lambda P, t, a, b: a * P - b * P**2
P_0 = 76.1
a = 0.02
b = 4e-5
t = np.linspace(0, 80, 9)

# Solucao numerica
P_n = odeint(d_P, P_0, t, args=(a, b))
# Solucao analitica
P_t = (89.7617 * np.exp(0.02*t))/(1 + 0.1795 * np.exp(0.02*t))

for i in range(len(t)):
    print(f"t = {t[i]}: P_n = {P_n[i][0]}, P_t = {P_t[i]}")
```

Tempo (t)	Valor Numérico (P_n)	Valor Teórico (P_t)
0.0	76.1000	76.1015
10.0	89.9187	89.9208
20.0	105.6215	105.6244
30.0	123.2424	123.2463
40.0	142.7389	142.7442
50.0	163.9775	163.9844
60.0	186.7244	186.7335
70.0	210.6488	210.6603
80.0	235.3357	235.3501

Tabela 3: Comparação entre os valores numéricos P_n e "exatos" P_t para $t \in [0, 80]$.

Analisando a tabela, é possível notar que os valores possuem diferença apenas a partir da casa centesimal, mostrando que a função `odeint` consegue gerar uma aproximação bem precisa do PVI.

Plotando a solução numérica do PVI com a função da solução "exata", na figura 2, podemos observar melhor como os pontos se aproximam bem da função:

```
legend_n = "Solucao Numerica (odeint)"
legend_t = rf"Solucao 'Exata' do PVI"
plt.scatter(t, P_n, label=legend_n, color="tab:blue", s=100)
plt.plot(t, P_t, label=legend_t, color="tab:orange")
plt.xlabel('t')
plt.ylabel('P')
plt.legend()
plt.show()
```

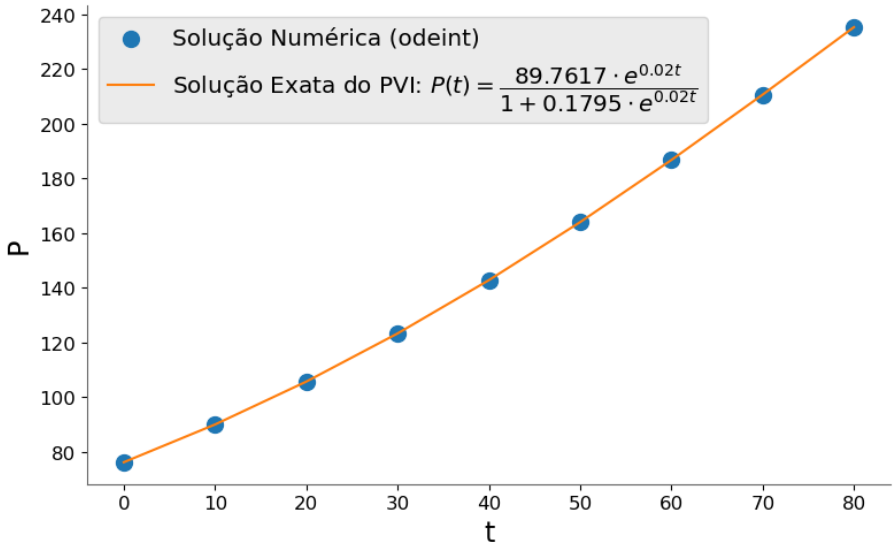

Figura 2: Solução numérica e "exata" para a equação diferencial dada.