Relatório de Cálculo Numérico

Projeto Computacional II

Professora: Kelly Cristina Poldi

Felipe Lobão Melara RA:247844
Gabriel Mattias Antunes RA:281199

Campinas, 19 de novembro de 2024

Introdução: Neste presente relatório, foram feitos diversos métodos numéricos para resolução de problemas, com o objetivo de integrar a parte teórica que presenciamos nas aulas da Professora Kelly, com o ambiente computacional, para que os alunos possam saber aplicar os conhecimentos obtidos, na prática.

Em relação a aplicação dos métodos, o relatório foi dividido em duas partes: a primeira parte era sobre a utilização de métodos para resolução de Problemas de Valor Inicial (PVI), utilizando o método de *Euler* aperfeiçoado e o método de *Runge-Kutta* de quarta ordem. A partir da aplicação de ambos os métodos, uma comparação é feita com a solução analítica do determinado problema, para que algumas comparações sobre precisão entre esses métodos pudessem ser feitas.

No que tange a segunda parte do presente relatório, uma curva de crescimento populacional de um certo país foi fornecida e para resolvê-la, foi utilizada a função *odeint* do *Python* e também, foi feita uma comparação com a solução analítica para fazer comparações sobre precisão.

Execução do Projeto Computacional:

Neste presente tópico, será feita a explicação do programa utilizado para a resolução de ambos problemas e a análise sobre a precisão de cada método.

Parte I:

Para a resolução da equação diferencial ordinária (EDO), primeiramente foi feito um truncamento, para evitar erros de precisão, a definição da EDO e também a importação das bibliotecas do Python utilizadas. Todas esses passos, foram realizados na **Figura 1**, encontrada abaixo:

```python
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# truncar um número por conta de erros de aproximação de float
# exemplo: na terceira iteração do loop nos métodos, em vez de x = 1.2, x = 1.2000000000000002
def truncar(num: float, casas_decimais: int) -> float:
    fator = 10 ** casas_decimais

    return math.trunc(num * fator) / fator

def funcao(x, y):
    return 1/(x**2) - y/x - y**2
```

Figura 1: Importação das bibliotecas, truncamento e definição da EDO.

A partir das definições iniciais, agora de fato os métodos serão aplicados, começando com o Euler aperfeiçoado, que se encontra na Figura 2, tanto sua definição no ambiente computacional, quanto sua aplicação.

```python
def euler_aperfeicoado(h: float, x_min: float, x_max: float, y_inicial:
    x = x_min
    y = y_inicial
    pontos_x = []
    pontos_y = [y_inicial]

    # aplica o método de Euler Aperfeiçoado
    while (x < x_max):
        k1 = funcao(x, y)
        k2 = funcao(x + h, y + h*k1)
        y = y + (h/2) * (k1 + k2) # fórmula do método
        pontos_y.append(y) # adiciona os pontos na lista
        pontos_x.append(x)
```

Figura 2: Definição e aplicação do Euler aperfeiçoado

Com o método de Euler aperfeiçoado pronto, o próximo passo do programa é definir e aplicar o método de Runge-Kutta de quarta ordem, que pode ser visualizado na Figura 3.

```python
def runge_kutta_ordem_4(h: float, x_min: float, x_max: float, y_inicial: float) -> list:
    x = x_min
    y = y_inicial
    pontos_y = [y_inicial]

    # aplica o método de Runge-Kutta de 4ª ordem
    while (x < x_max):
        k1 = funcao(x, y)
        k2 = funcao(x + h/2, y + h*k1/2)
        k3 = funcao(x + h/2, y + h*k2/2)
        k4 = funcao(x + h, y + h*k3)
        y = y + (h/6) * (k1 + 2*k2 + 2*k3 + k4) # fórmula do método
        pontos_y.append(y) # adiciona os pontos na lista
        x = truncar(x + h, 1)

    return pontos_y
```

Figura 3: Aplicação e definição do método de Runge-Kutta

Para finalizar a parte das equações, falta gerar os pontos da solução analítica, para que a comparação final possa ser feita:

```python
# gera os pontos da solução analítica para colocar no gráfico
def pontos_analiticos(x_min, x_max) -> list:
    h = 0.0001
    x = x_min
    pontos_x = []
    pontos_y = []

    while (x <= x_max):
        y = -1/x
        pontos_x.append(x)
        pontos_y.append(y)
        x += h

    return pontos_x, pontos_y
```

Figura 4: Criação dos pontos da solução analítica

Como passo final do programa, foi realizada a plotagem de todas as soluções, Euler aperfeiçoado, Runge-Kutta de quarta ordem e a solução analítica, para em seguida, a comparação ser feita.

```python
# faz o gráfico dos três métodos e o imprime na tela
def grafico(x: list, y_euler: list, y_rk: list, x_analitico: list, y_analitico: list):
    fig, varx = plt.subplots()
    varx.plot(x, y_euler, label = 'Euler Aperfeiçoado')
    varx.plot(x, y_rk, label = 'Runge-Kutta de 4ª ordem')
    varx.plot(x_analitico, y_analitico, label = 'Solução Analítica')
    varx.set_xlabel('Pontos xi')
    varx.set_ylabel('Aproximações yi')
    varx.set_title('Pontos xi x Aproximações yi')
    varx.legend()
    plt.show()

pontos_xi, pontos_yi_euler = euler_aperfeicoado(0.1, 1.0, 2.0, -1.0)
print(f'Aproximações por Euler Aperfeiçoado: {pontos_yi_euler}')
print('')

pontos_yi_rk = runge_kutta_ordem_4(0.1, 1.0, 2.0, -1.0)
print(f'Aproximações por Runge-Kutta de 4ª ordem: {pontos_yi_rk}')

pontos_xi_analiticos, pontos_yi_analiticos = pontos_analiticos(1.0, 2.0)
grafico(pontos_xi, pontos_yi_euler, pontos_yi_rk, pontos_xi_analiticos, pontos_yi_analiticos)

# Fim da Parte 1
```

Figura 5: Plotagem do gráfico com todas soluções

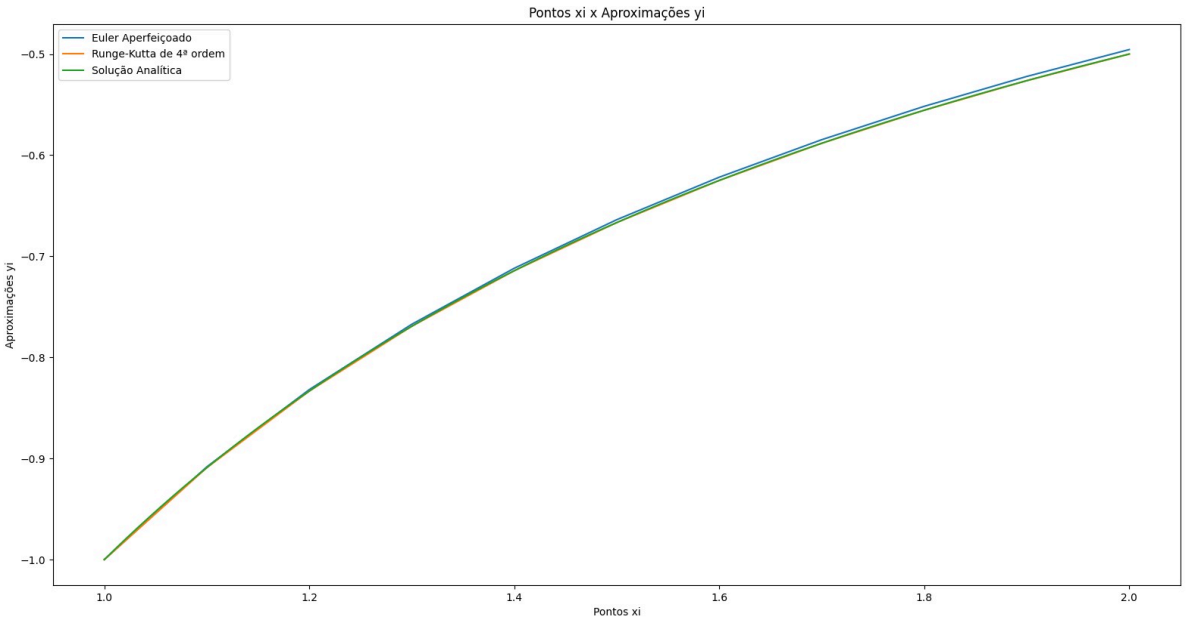

Figura 6: Gráfico das soluções obtidas (Pontos xi X Aproximação yi)

Analisando a Figura 6, podemos concluir que ambos métodos geraram soluções semelhantes, dado que é difícil diferenciá-las no gráfico e portanto, afirmamos que os métodos são eficazes. Porém, o método de Euler aperfeiçoado (curva azul), teve uma maior diferenciação da solução analítica (curva verde), pois é perceptível as linhas se “separando”, já o Runge-Kutta de quarta ordem (linha amarela), quase não é perceptível a diferença com a solução analítica, mostrando uma maior precisão.

Parte II:

Para a resolução da parte II, foi utilizada a função *odeint* do *Python,* para resolução de EDO's. Foi gerada uma tabela para comparação dos valores da solução analítica e da solução gerada pelo *odeint.*

Iniciando o código da parte II, é gerada uma tabela no Excel, contendo a solução gerada pelo odeint e a solução exata, em cada ponto analisado.

```
# faz a tabela comparativa dos pontos
def tabela(lista1: list, lista2: list, lista3: list):
    table = pd.DataFrame()
    table['t'] = lista1
    table['P(t) - odeint'] = lista2
    table['P(t) - sol. exata'] = lista3
    print(table)
    table.to_excel('Tabela_Parte2_PCII.xlsx', index = False)

# define uma lista contendo os pontos da solução exata calculada em cada t
def sol_exata() -> list:
    pontos = []
    for i in range(9):
        t = i * 10
        e = math.e
        pontos.append((89.7617 * (e**(0.02*t))) / (1 + 0.1795 * (e**(0.02*t))))

    return pontos
```

Figura 7 - Criação da tabela comparativa das soluções

A partir disso, basta utilizar o odeint para fazer o cálculo aproximado da solução, para que ambas soluções sejam comparadas no final do relatório. Importante ressaltar que foi feito um arredondamento para não gerar frações, pois se trata de uma curva habitacional. A derivada também foi calculada no código.

Como citado anteriormente, este código gera uma tabela, que pode ser encontrada na Figura 9.

```
# retorna a derivada de P
def model(P, t):
    a = 0.02
    b = 0.00004

    dPdt = a*P - b*(P**2)

    return dPdt

# calcula a EDO utilizando odeint() e faz a tabela de pontos comparando os dois métodos
def edo(P_inicial: float):
    t = np.linspace(0, 80, 9)
    P = odeint(model, P_inicial, t)
    exata = sol_exata()

    for i in range(len(P)):
        P[i] = truncar(float(P[i]), 6) # não tem como ter frações de habitantes

    tabela(t.tolist(), P, exata)

edo(76.1)
```

Figura 8 - utilização do odeint

t	P(t) - odeint	P(t) - sol. exata	Erro absoluto
0	76,100000	76,101484	0,00148368
10	89,918712	89,920792	0,002080085
20	105,621483	105,624362	0,002878791
30	123,242356	123,246280	0,003924234
40	142,738949	142,744220	0,005271006
50	163,977455	163,984420	0,00696467
60	186,724437	186,733474	0,009036746
70	210,648762	210,660267	0,01150455
80	235,335710	235,350077	0,014367321

Figura 9 - Tabela comparativa da solução gerada pelo odeint e a solução exata, com o erro absoluto.

Analisando a Figura 9, é possível concluir que o método *Odeint*, foi muito preciso em seus resultados, pois, a solução exata se assemelha bastante com os valores gerados pelo método (maioria dos erros na terceira casa decimal).