UNIVERSIDADE DO ESTADO
DO RIO DE JANEIRO

INSTITUTO POLITÉCNICO
GRADUAÇÃO EM
ENGENHARIA DA COMPUTAÇÃO

Vitor Saraiva
Gabriel Calheias

Métodos Numéricos para Equações Diferenciais
Trabalho 1

Nova Friburgo
2022

As etapas do trabalho referentes às modelagem computacionais dos métodos de **Runge-Kutta de 3ª Ordem** e de Problema de Contorno, utilizando-se de **Jacobi**, foram executadas na linguagem de programação **Python**.

As bibliotecas utilizadas neste trabalho são **matplotlib** para o plot de gráficos além de **math**, **numpy** e **scipy** para manipulações aritméticas. Ademais, utilizou-se a biblioteca **pandas**, na Parte II, para a construção de dataframes.

Parte I

A parte 1 do trabalho foi dedicada ao encontro de aproximações numéricas para as equações de concentração de três reagentes. Em **Python**, para a realização destas codificações com as seguintes fórmulas:

$$\frac{dc_a}{dt} = -\alpha c_a c_c + c_b \,,$$

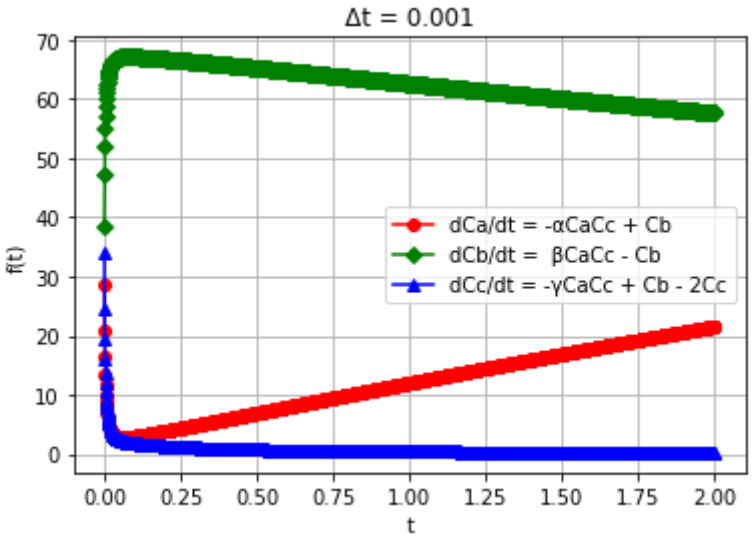
$$\frac{dc_b}{dt} = \beta c_a c_c - c_b \,,$$

$$\frac{dc_c}{dt} = -\gamma c_a c_c + c_b - 2c_c$$

Com as condições iniciais pré-estabelecidas nas orientações do trabalho para $c_b = 0$ e $t = 0$, inicial, até um t máximo.

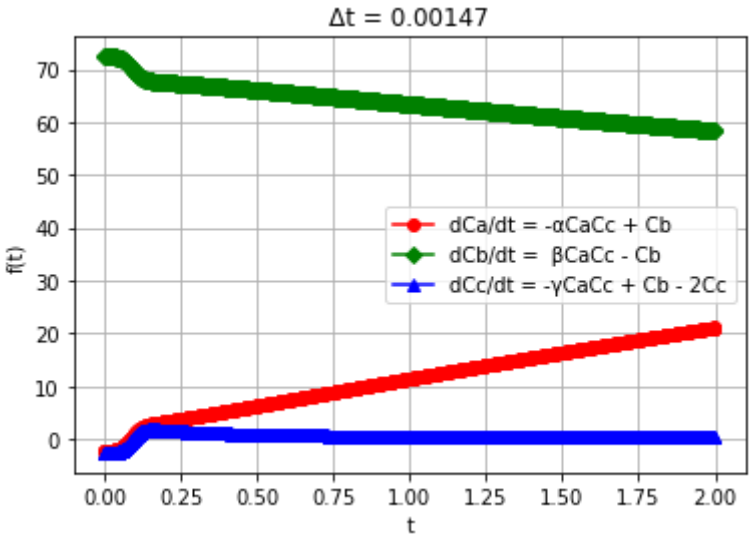
Aplicando valores facultativos, seguindo determinada regra, para as demais variáveis pode-se utilizar do método de **Runge-Kutta de 3ª Ordem** para obter as aproximações numéricas.

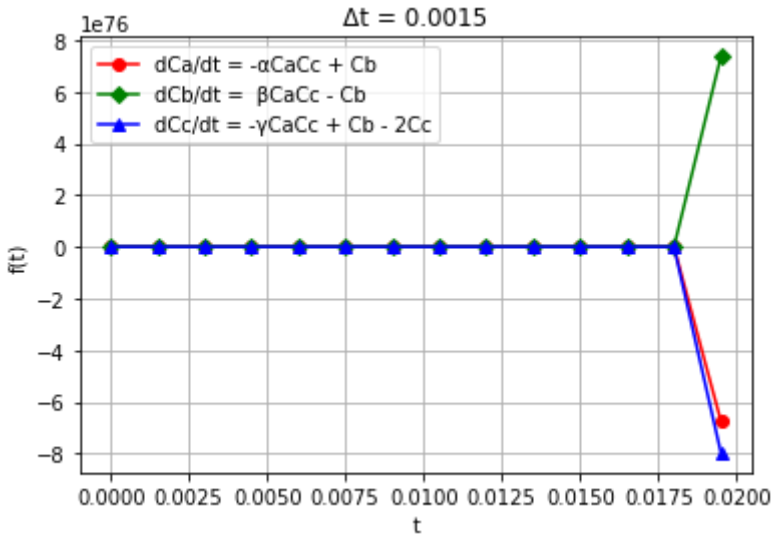
Para α, β e γ, cujos valores são facultativos em torno de uma dezena, foram escolhidos inicialmente 11, 12 e 13, respectivamente. Já, para os valores iniciais de c_a e c_c, facultativos por volta de algumas dezenas, escolheu-se 64 e 76, respectivamente. Para o t máximo, foi escolhido 2 segundos, com o $\Delta t = 0.001$.



O Δt foi modificado algumas vezes e, destas modificações, constatou-se que fora de um intervalo, os valores da função não mais respeitavam o comportamento esperado.

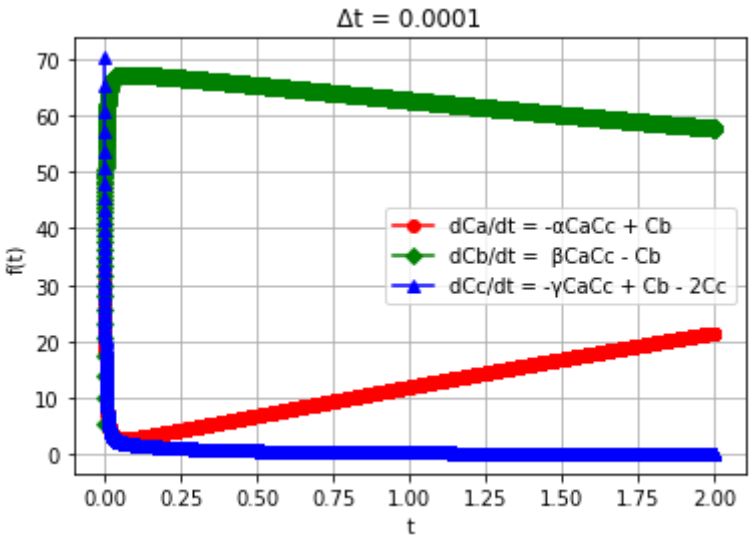
Superiormente, valores para Δt acima de 0.00147 influíram em resposta "nan" (not a number) nas etapas do Runge-Kutta, respeitando um estudo de 5 casas decimais após a vírgula.

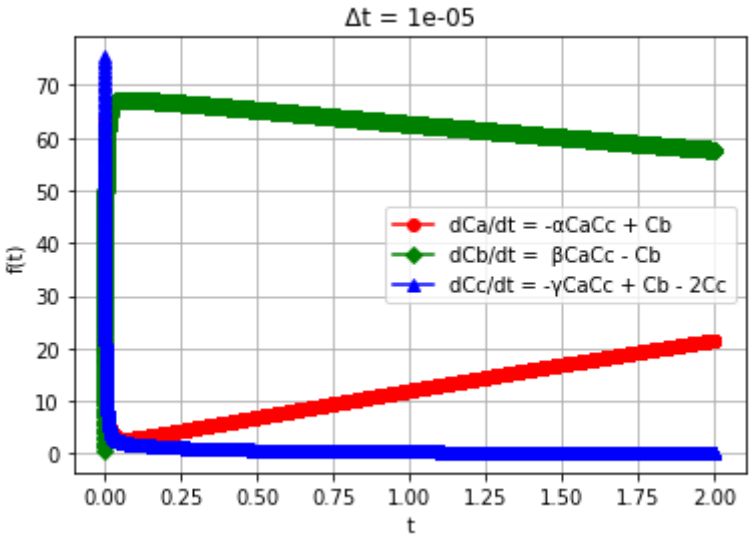




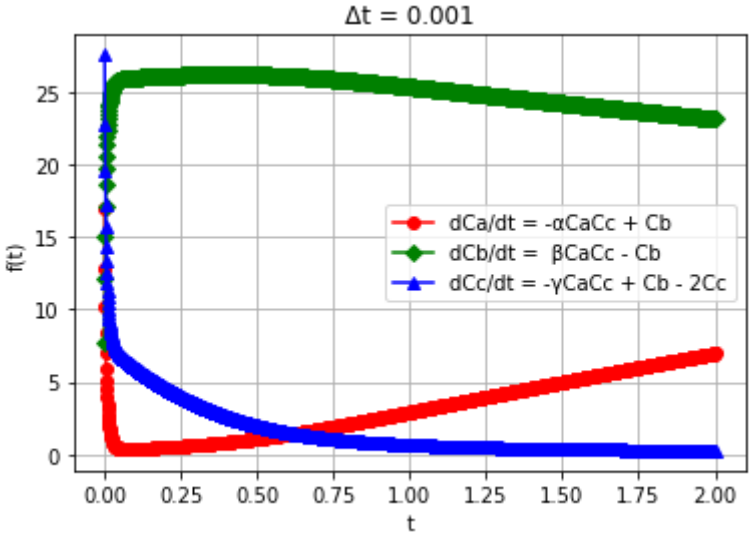
Inferiormente, testes com valores menores (e até muito menores) para Δt foram realizados, mas com o adendo de que o tempo de execução aumentava consideravelmente, embora seja sabido que a solução, neste processo, esteja sendo refinada.

Com valores na faixa de 10^{-3} a resposta do programa era quase instantânea, enquanto que para 10^{-4}, o programa demorava cerca de 40 segundos para ser completamente executado. Para valores de 10^{-5}, a execução custou por volta de 5 minutos. E, para valores na faixa de 10^{-6}, foram necessários mais de 35 minutos para completar a execução.





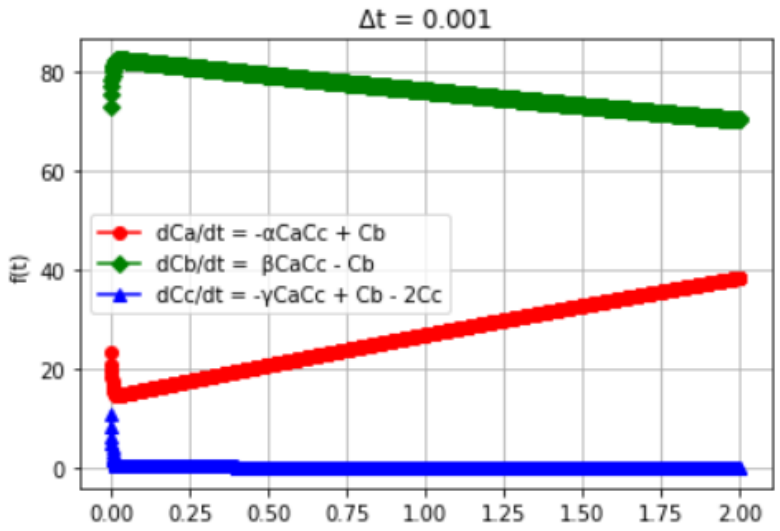
Mantendo α, β e γ, escolhidos inicialmente como 11, 12 e 13, respectivamente e alterando os valores iniciais de c_a e c_c, para 24 e 36 e utilizando o t máximo como 2 segundos, com o Δt = 0.001, a curva sofre uma suavizada em relação à anterior e o Δt máximo muda conforme mostrado no gráfico abaixo onde é possível utilizar ele como 0.002.



Observou-se que quanto maior os parâmetros c_a e c_c (muito grandes no exemplo abaixo, 90 para ambos), o comportamento do gráfico se assemelha visualmente ao linear (entretanto, o ponto de vista é, evidentemente, influenciado por uma mudança de escala do eixo da ordenada) e que, principalmente para o parâmetro c_a, os resultados se

traduzem em grandes variações (resultados final "–" inicial) na ordenada.

Ao passo que, quanto menores os parâmetros de c_a e c_c, mais suaves e aproximados de curvas são os comportamentos do gráfico (novamente, pontuando que é indubitável considerar que a mudança de escala influencie, mesmo que ligeiramente, estas percepções) como expresso no gráfico acima.



.

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

fCa = []
fCb = []
fCc = []
tempo = []

# Por volta de uma dezena
alfa = 11
beta = 12
gamma = 13

# Por volta de algumas dezenas
A = 64
B = 0
C = 76

t = 0
```

```
# Por volta de algumas unidades
tf = 2

# Passo de Tempo
# Max Calculado = 0.0014 ----- 0.00148 os valores de concentração se tornam nan
Δt = 0.001


def dca(ca, cb, cc): return -alfa*ca*cc + cb
def dcb(ca, cb, cc): return beta*ca*cc - cb
def dcc(ca, cb, cc): return -gamma*ca*cc + cb - 2*cc


def RungeKutta(A, B, C, t, tf, Δt):
    fa = A
    fb = B
    fc = C
    while (t < tf):
        print("\nt = " + str(t))

        ka1 = dca(fa, fb, fc)
        kb1 = dcb(fa, fb, fc)
        kc1 = dcc(fa, fb, fc)

             ka2  =  dca(fa  +  (ka1*(Δt/2)),  fb  +  (kb1*(Δt/2)),  fc  +
(kc1*(Δt/2)))
             kb2  =  dcb(fa  +  (ka1*(Δt/2)),  fb  +  (kb1*(Δt/2)),  fc  +
(kc1*(Δt/2)))
             kc2  =  dcc(fa  +  (ka1*(Δt/2)),  fb  +  (kb1*(Δt/2)),  fc  +
(kc1*(Δt/2)))

            ka3  =  dca(fa  -  (ka1*Δt)  +  (2*ka2*Δt),  fb  -  (kb1*Δt)  +
(2*kb2*Δt), fc - (kc1*Δt) + (2*kc2*Δt))
            kb3  =  dcb(fa  -  (ka1*Δt)  +  (2*ka2*Δt),  fb  -  (kb1*Δt)  +
(2*kb2*Δt), fc - (kc1*Δt) + (2*kc2*Δt))
            kc3  =  dcc(fa  -  (ka1*Δt)  +  (2*ka2*Δt),  fb  -  (kb1*Δt)  +
(2*kb2*Δt), fc - (kc1*Δt) + (2*kc2*Δt))

        ka = (Δt*(ka1+(4*ka2)+ka3))/6
        kb = (Δt*(kb1+(4*kb2)+kb3))/6
        kc = (Δt*(kc1+(4*kc2)+kc3))/6
```

```
        fa = fa + ka
        fb = fb + kb
        fc = fc + kc

        print("RK3 dCa = " + str(fa))
        print("RK3 dCb = " + str(fb))
        print("RK3 dCc = " + str(fc))

        fCa.append(fa)
        fCb.append(fb)
        fCc.append(fc)
        tempo.append(t)

        t = t + Δt


print("Alpha = " + str(alfa))
print("Beta = " + str(beta))
print("Gamma = " + str(gamma))
print("\nA = " + str(A))
print("B = " + str(B))
print("C = " + str(C))
print("Δt = " + str(Δt))
RungeKutta(A, B, C, t, tf, Δt)

plt.plot(tempo, fCa, color = 'red', marker = 'o', label='dCa/dt =
-αCaCc + Cb')
plt.plot(tempo, fCb, color = 'green', marker = 'D', label='dCb/dt =
βCaCc - Cb')
plt.plot(tempo, fCc, color = 'blue', marker = '^', label='dCc/dt =
-γCaCc + Cb - 2Cc')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.grid(True)
plt.title('Δt = ' + str(Δt))
plt.legend()
```

Parte II

Para a segunda parte do trabalho, implementou-se o método de Jacobi para resolução de sistemas de equações.

A equação de concentração referente a Parte II é:

$$D\frac{d^2C}{dx^2} - kC = 0$$

O Δx empregado foi de 0.05 em um tubo de comprimento L = 20 (em que L deve ser de algumas dezenas). Para os termos α (de algumas unidades) e β (de 2 a 3 vezes maior que α) foram escolhidos 5 e 3 * 5 = 15, respectivamente.

```
import matplotlib.pyplot as plt
import numpy as np
import math
import scipy
import pandas as pd

Δx = 0.5
Lmax = 20
i = int(Lmax/Δx)
nofinal = i-1
print(i)

L = []
count = 0
while(count < Lmax):
  L.append(count)
  count += Δx

#print(L)

C = [0]*(i+1)
A = np.zeros((i-2,i-2),dtype=np.float64)
b = np.zeros((i,1),dtype=np.float64)

C[1] = 0.1
C[i] = 0

alpha = 5
D = alpha*10**-6

beta = 3*alpha
k = beta*10**-6
```

```
def criterio_linhas(A):
    swap = np.copy(A)
    b = np.diag(A)
    A = A - np.diagflat(b)
    x = np.ones(b.size)
    permutation = b.size**2
    acc = True

    while(acc and permutation > 0):
        for i in range(b.size):
            x[i] = np.sum(A[i])/b[i]

        if(np.amax(x) < 1): acc = False
        else:
            permutation = permutation-1
            swap = np.random.permutation(swap)
            A = np.copy(swap)
            b = np.diag(A)
            A = A - np.diagflat(b)

    return np.amax(x)

#Matriz A, Matriz b dos termos independentes e N o número de iterações
e o erro
def jacobi(A, b, N, chute, erro = 0.001):
    if(criterio_linhas(A) > 1):
        print("O sistema não converge para o método de Jacobi")
        return

    x = np.diag(A) #recebe um vetor contendo a diagonal principal
    A = A - np.diagflat(x) #Zera a diagonal principal de A

        #Para dividir todos os valores da matriz A pelos termos
independentes
    for i in range(x.size):
        A[i] = A[i]/x[i]
        b[i] = b[i]/x[i]

    x = np.copy(chute)
    swap = np.zeros(x.size)

    A = A*-1
```

```
    for stop in range(N):
        for i in range(x.size):
            swap[i] = np.sum((A[i]*x))+(b[i])
        #Cálculo da tolerância ou erro
        data_df_jacobi = pd.DataFrame(swap)
        #print(f"Iteração {stop}: {swap}")
        print("\nIteração ",stop)
        print(data_df_jacobi)
        if((np.linalg.norm(swap) - np.linalg.norm(x)) < erro):
          print("\nFinalizou por Erro\n")
          return swap
        x = np.copy(swap)

    return x

def F(Ci):
  j = count-2
  if(Ci == 2):
    #print("i = 2")
    A[Ci-2,j] = ((2*D)+((Δx**2)*k))
    A[Ci-2,j+1] = -D
    b[Ci-2,0] = D*C[1]
    #print(A[0,:])
  if(Ci == nofinal):
    #print("i = L")
    A[Ci-2,j-1] = -D
    A[Ci-2,j] = ((2*D)+((Δx**2)*k))
    b[Ci-2,0] = D*C[i]
    #print(A[nofinal-2,:])
  if((Ci > 2) & (Ci < nofinal)):
    #print("2 < i < L")
    A[Ci-2,j-1] = -D
    A[Ci-2,j] = ((2*D)+((Δx**2)*k))
    A[Ci-2,j+1] = -D
    b[Ci-2,0] = 0
    #print(A[Ci-2,:])

count = 2
while count <= nofinal:
  F(count)
  count += 1
```

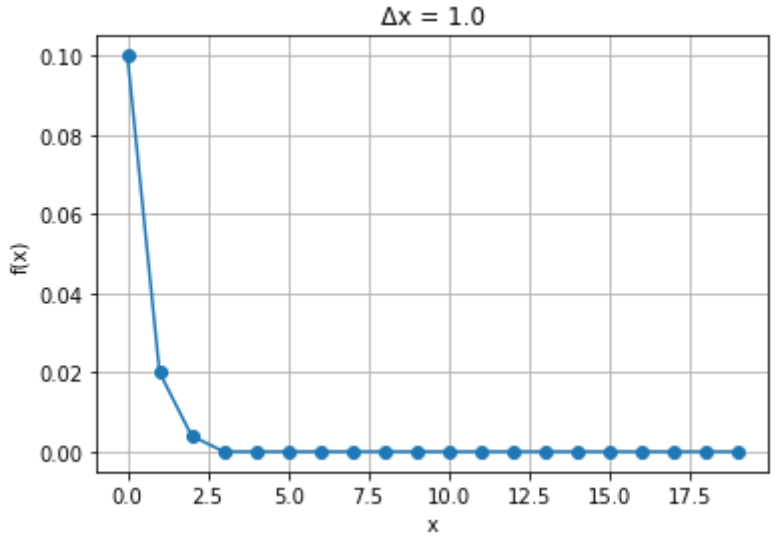
```
data_df_A = pd.DataFrame(A)
data_df_b = pd.DataFrame(b)
print("\nMatriz A:\n",data_df_A)
print("\nVetor b:\n",data_df_b,"\n")

'''print("\nMatriz A:\n",A)
print("\nVetor b:\n",b,"\n")'''

x = np.zeros(i-2)

print("Jacobi:")
r = jacobi(A, b, 100, x)
data_df_r = pd.DataFrame(r)
print("\nSolução:\n",data_df_r,"\n")

'''count = 2
while count <= nofinal:
  print("C" + str(count) + " = " + str(r[count-2]))
  count += 1'''

r_list = []
r_list.append(0.1)
count = 0
while(count < i-2):
  r_list.append(r[count])
  count += 1
r_list.append(0)


plt.plot(L, r_list, marker = 'o')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
plt.title('Δx = ' + str(Δx))
```

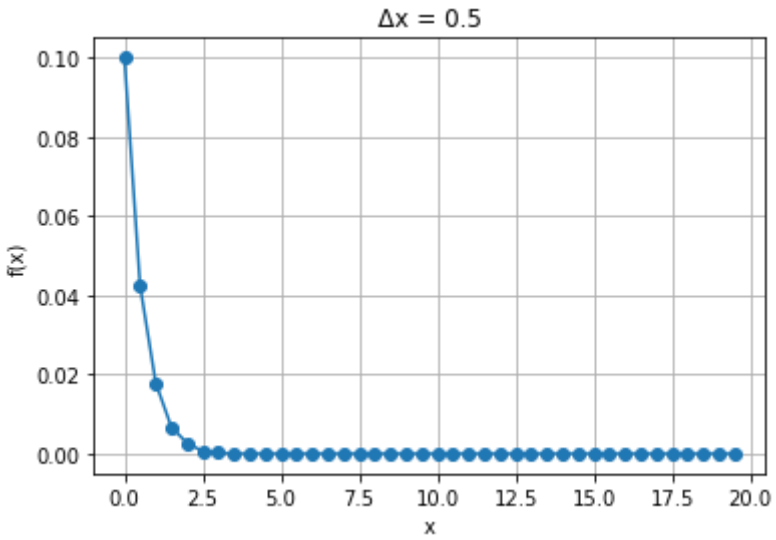
Através da fórmula:

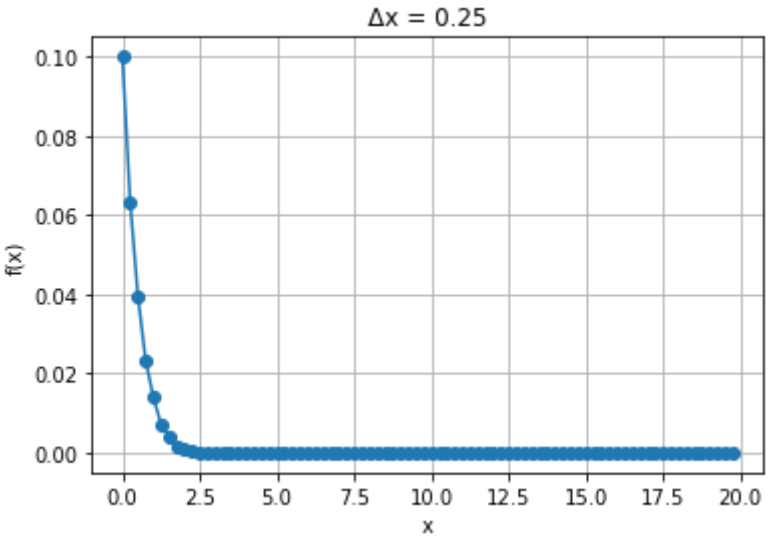
$$i = int\left(\frac{L_{max}}{\Delta x}\right)$$

Em que i é o número de nós na malha, atesta-se que quanto menor o Δx e/ou maior o L_{max} o número de nós aumenta.

Desta forma, ao passo que o parâmetro i (número de nós) aumenta, refina-se as soluções, porém exige-se muito poderio computacional e muito tempo para o resultado ser apresentado. Se diminuirmos cada vez mais o Δx, o número de nós aumenta com relação a fórmula acima. Fixando os Δx para os mostrados nos títulos dos gráficos e o L_{max} em 20 temos:





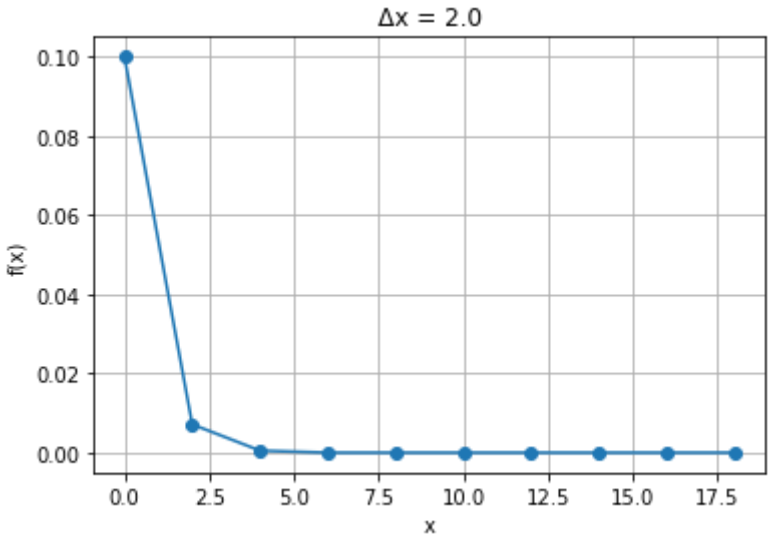


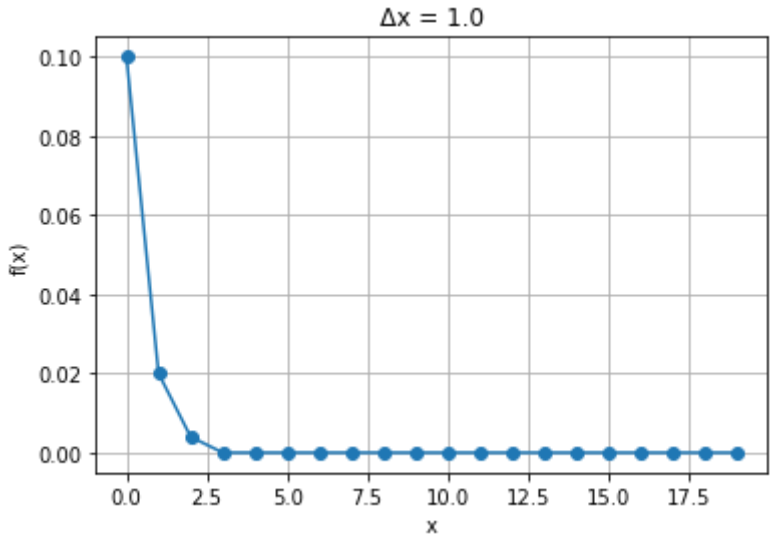
Agora, modificando os parâmetros e fixando o número de nós, encontra-se Δx ou o L_{max} através da fórmula:

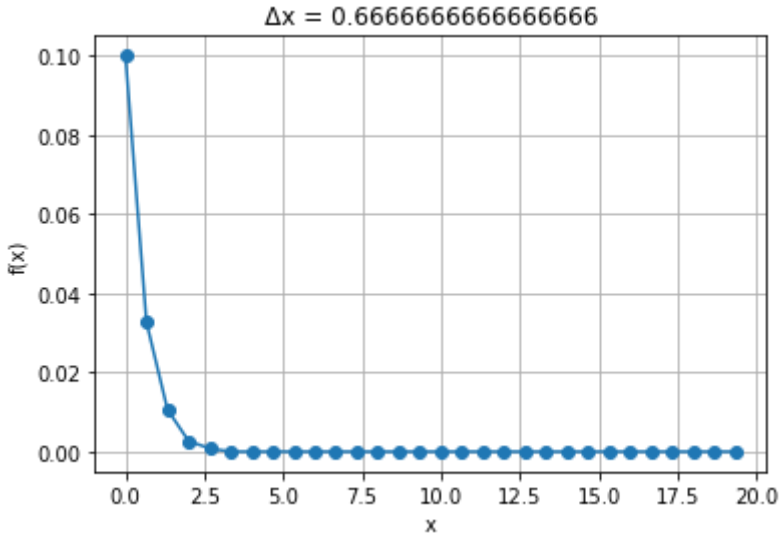
$$\Delta x = \frac{L_{max}}{i} \quad ,$$

$$L_{max} = \Delta x * i$$

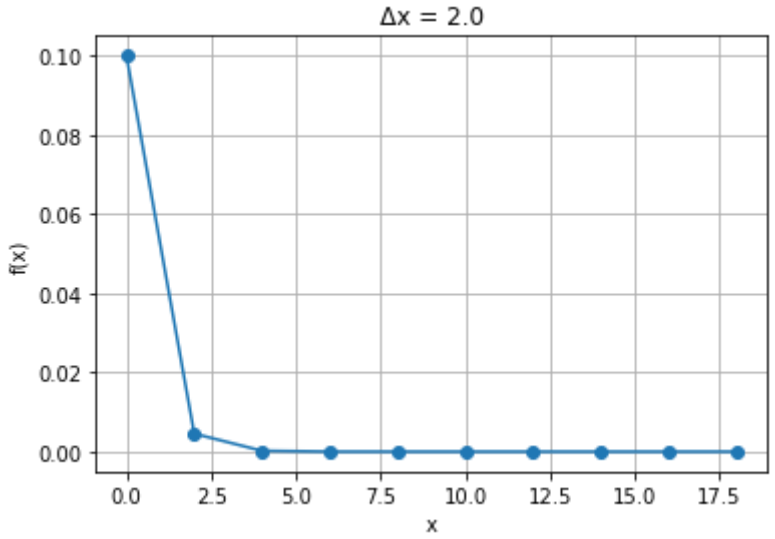
Fixando o L_{max} em 20 e modificando o número de nós para 10, 20 e 30 respectivamente, tem-se:

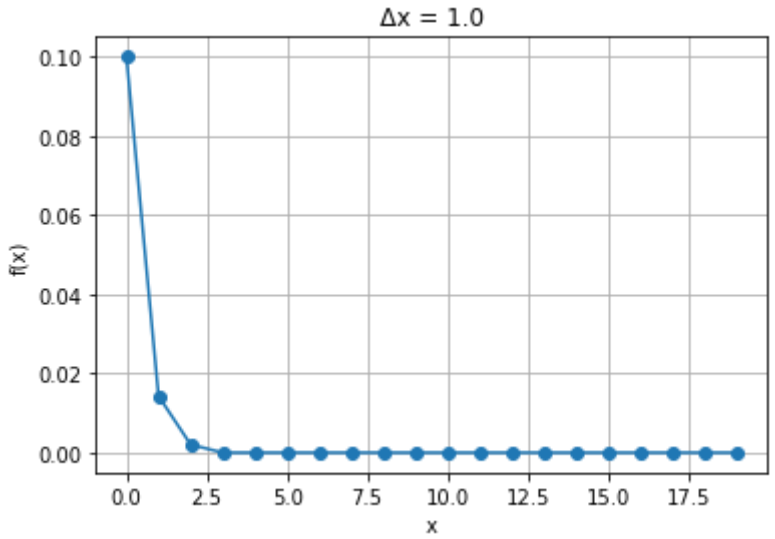


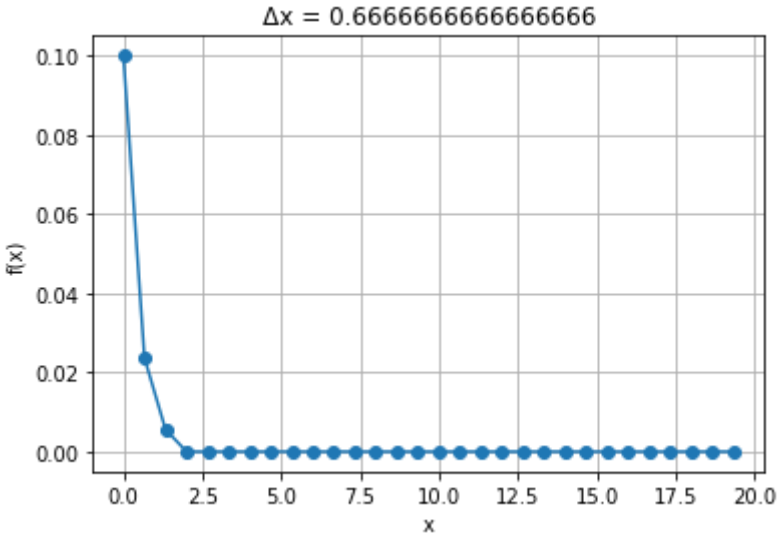




Agora modificando o α para 7 e β cinco vezes α, utilizando também os parâmetros de nós e L_{max} anteriores, obtêm-se os gráficos:




Δx = 1.0
0.10
0.08
0.06
0.04
0.02
0.00
f(x)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
x


Δx = 0.6666666666666666
0.10
0.08
0.06
0.04
0.02
0.00
f(x)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
x

Bibliografia

- Material Didático da disciplina disponibilizado na plataforma Moodle
- Orientações do Trabalho por aulas e pelo arquivo disponibilizado
- TCC "AVALIAÇÃO DAS SOLUÇÕES ANALÍTICA E NUMÉRICA DO PERFIL DE CONCENTRAÇÃO EM UM CATALISADOR ESFÉRICO" de Juliana Meirelles de Almeida Souza, Leandro Vidal Schot e Roberta Cardoso Garcia de Freitas Gama, da Graduação em Engenharia Química, da UFF em 2017

Google Collab

- https://colab.research.google.com/drive/1xQ204slX8K36VkxrijPJYXyYeDvfR2zr?usp=sharing#scrollTo=k1iutxM1L815