**TRABALHO COMPUTACIONAL 5**

Nomes:

* Daiana Santos 120.357
* Isadora Muniz 120.431
* Luciana Bello 120.506
* Maria Victória Siqueira 120.529

# 

# **Parte 1**

## **Exercício 1**

A regra do trapézio é dada por:

, onde e

No exercício foi dado:

N = 30

a = -2

b = 5

Assim,

Portanto, pela regra dos trapézios:

## **Exercício 2**

**Regra dos trapézios**

A regra do trapézio é dada por:

, onde e

**a)**

Segundo o exercício:

a = 1

b = 2

Deve utilizar 4 e 6 divisões,

iniciando com N=4:

**4,6951**

com N=6:

**4,6816**

**b)**

Segundo o exercício:

a = 1

b = 4

Deve utilizar 4 e 6 divisões,

iniciando com N=4:

**4,6551**

com N=6:

**4,6615**

**c)**

Segundo o exercício:

a = 2

b = 14

Deve utilizar 4 e 6 divisões,

iniciando com N=4:

**4,7684**

com N=6:

**4,7078**

**Regra de Simpson**

A regra de Simpson simples de acordo com a integral é dada por:

;

A fim de melhorar a precisão do valor esperado, é feita a divisão em mais partes o intervalo da integral, o que será aplicado nos exercícios.

1. **Divisão em 4 partes:**

= 0,25

**Divisão em 6 partes:**

= 0,1666

1. **Divisão em 4 partes:**

= 0,75

**Divisão em 6 partes:**

= 0,5

= 4,6665

1. **Divisão em 4 partes:**

= 3

**Divisão em 6 partes:**

= 2

**Comparação entre os dois**

|  | **Valor exato** | **Regra dos trapézios - n=4** | **Regra dos trapézios - n=6** | **Regra de Simpson - n=4** | **Regra de Simpson - n=6** |
| --- | --- | --- | --- | --- | --- |
| **Exer a)** | 4,67077 | 4,6951 | 4,6816 | 4,6708 | 4,6707 |
| **Exer b)** | 4,6666 | 4,6551 | 4,6615 | 4,6662 | 4,6665 |
| **Exer c)** | 4.65488 | 4,7684 | 4,7078 | 4,6763 | 4,6614 |

De acordo com os cálculos é possível observar que os valores das integrais calculados pela Regra de Simpson obtiveram resultados muito próximos do valor exato, sendo assim um método de maior eficiência quando comparado à Regra dos trapézios.

## **Exercício 3 - Obtenção de erros menores que**

Com base nos resultados obtidos no exercício anterior, os valores das integrais calculados pelos dois métodos com uma divisão de 6 partes do intervalo de integração, resultou em um erro entre , sendo assim, cálculos com divisões de pelo menos 10 partes do intervalo de integração resultarão em erros menores que .

## **Exercício 4**

**a) Regra de Simpson**

A regra de Simpson simples de acordo com a integral é dada por:

;

Para maior precisão, será feito um cálculo considerando uma divisão de 4 partes do intervalo de integração.

**Divisão em 4 partes:**

= 0,15

**b) Regra dos trapézios**

A regra do trapézio é dada por:

, onde e

Segundo o exercício:

a = 0

b = 0,6

Como o exercício não forneceu a quantidade de subdivisões, considerando N=5:

**0,471**

# 

## **Exercício 5**

A regra de Simpson simples de acordo com a integral é dada por:

;

Realizando o cálculo com a regra de Simpson simples para a integral temos:

= /4

**Erro** = |1 - 1,0022| = 2,279 x

**Divisão em 4 partes:**

= /8

=1,000

**Erro =** |1 - 1,00013| = 1,345 x

Logo, h = /8 e a divisão deve ir de x1 a x6 para que o erro da integral seja menor que .

# **Parte 2**

## 

## **Exercício 1**

1. dM/dt = - k M(t)
2. dM/M(t) = − k.dt

∫ dM/M(t) = ∫ - k .dt

ln(M(t)) = − k.t + c

M(t) = = .

M(t) =.

Para t=0,

M(0) = C =

M(t) = .

1. M(t) = .

O tempo de meia-vida é estabelecido quando M equivale a metade de , logo:

½ . = .

½ =

ln ½ =

ln ½ =

k =

1. ¼ . = .

¼ =

ln ¼ =

t = 11466,45 anos

1. 0,3 . = .

0,3 =

ln 0,3 =

t = 9958,42 anos

## **Exercício 2**

**Valores :**

t0 = 0

t1 = 2hT = 20°C

Para encontrar o valor da constante k, utiliza-se a seguinte expressão:

θ1 − T = (θ0 − T) exp (−κ t1)

Temos:

= exp (−κ t1) ; aplicando logaritmo :

⇒

Considerando que θm seja a temperatura do corpo humano e tm a hora da morte, temos a seguinte equação:

θm = T + (θ0 − T) exp (−κ tm)

A fim de encontrar ™, aplica-se o logaritmo na equação

⇒

Ou seja, é possível determinar a hora do crime, que no caso foi há um pouco mais de uma hora antes do momento em que o corpo foi encontrado.

# **Parte 3**

## **Exercício 1**

O **Método de Euler** é um método simples utilizado para resolver problemas de valor inicial, tendo como formato a equação:



sendo h o passo fixo dentro do intervalo.

Dado a partir do enunciado que **h=0,01** e que o PVI é **y’=y y(0)=1 temos que: Y(0,4)= 1,488864**.

## **Exercício 2**

Ainda utilizando o **Método de Euler**, com **h= 0.2** e tendo o PVI como **y’=4x³; y(0)=0** temos que **y(1)= 0,6400000**

## **Exercício 3**

Ainda utilizando o **Método de Euler**, tendo o PVI como **xy’= x-y; y(2)=2** temos que para **h=0.1**, y**(2.1)= 2,00000**

Para **h=0.01**, temos **y(2.1)= 2,002153110048**

Por fim, para **h= 0,025**, temos **y(2.1)=2,001807228916**

## **Exercício 4**

**a)** Utilizando o **Método de Euler** para a EDO com PVI **y’= 0,04y; y(0)=1000** para **h= 0,5** temos que **y(1)=1040,400000**

Para **h=0,25** temos **y(1)=1040,60401000000**

Para **h=0,1** temos **y(1)=1040,727734018910**

**b)** Através do **Método de Runge- Kutta de Ordem 2**, para resolver o PVI acima foi:

Para **h=0,5** temos **y(1)=1040,808040**

Para **h=0,25** temos **y(1)=1040,81005**

Para **h=0,1** temos **y(1)=1040,810664**

**c)** Usando o **Método de Runge-Kutta** **de Ordem 4** para resolver o problema, os resultados obtidos para h = 0.5, h = 0.25 e h = 0.1 foram:

| **h = 0.5** | 1000.0000 | 1020.201340 | **1040.810774** |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **h = 0.25** | 1000.0000 | 1010.050167 | 120.201340 | 1030.454534 | **1040.810774** |  |  |  |  |  |  |
| **h = 0.1** | 1000.0000 | 1004.008011 | 1008.032086 | 1012.072289 | 1016.128685 | 1020.201340 | 1024.290318 | 1028.395684 | 1032.517505 | 1036.655846 | **1040.810774** |

Os resultados foram analisados a partir de **x = 0** até **x = 1** para podermos observar as aproximações feitas a cada iteração do programa. Dessa forma, o resultado pedido pelo enunciado foi o equivalente a **y(1)** obtendo o valor de **1040.810774**.

## **Exercício 5**

Neste exercício, a EDO inicial foi substituída por ( de forma que **y(1) = 1.5**. Sendo assim, o problema pede para que encontremos o resultado de **y(1.6)** usando **h = 0.2** e **h = 0.1**, logo:

| **x** | **1.0** | **1.1** | **1.2** | **1.3** | **1.4** | **1.5** | **1.6** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **h = 0.1** | 1.5000000 | 1.742747 | 2.237705 | 3.175322 | 4.979302 | 8.627938 | **16.517450** |
| **h = 0.2** | 1.5000000 |  | 2.024700 |  | 3.337381 |  | **6.714698** |

Desse modo, os resultados para y(1.6) para h = 0.1 e h = 0.2 foram **16.517450** e **6.714698**, respectivamente.

## **Exercício 6**

**a)** Para obter o valor de K analiticamente foi resolvida a EDO apresentada no problema. A equação por ser linear, separável e homogênea pôde ser resolvida pelo método das equações separáveis, logo:

Integrando dos dois lados:

Sabendo que em t = 0 a concentração do remédio será a mesma da inicial(), podemos substituir na equação encontrada para descobrir o valor da constante P:

Logo, o valor de P é igual ao da concentração inicial do remédio. Sendo assim, podemos aplicar os valores iniciais dado pelo problema para descobrir o valor de k. Assim:

Utilizando os dados t = 4 dias e C(4) = 0,5:

=

O valor da constante K é **0.173286** de acordo com resolução da EDO acima.

**b)** Sabendo o valor da constante K, podemos substituir na equação para descobrir o valor do tempo quando a concentração do remédio for de 10%(o equivalente a ter eliminado os 90% do remédio). Para conseguir obter uma estimativa de resultado, usamos = 10 unidades de concentração. Logo, aplicando o **Método de Runge-Kutta** ao problema com **h = 1.0**, obtivemos os seguintes resultados:

| **Dias** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **h = 1.0** | 10.000000 | 8.408983 | 7.071100 | 5.946077 | 5.000045 | 4.204530 | 3.535583 | 2.973066 | 2.500046 | 2.102285 | **1.767808** |

Assim, o remédio teria sido eliminado em 90% aproximadamente no **10º dia**.

## **Exercício 7**

**a)** Usando o **método de Runge- Kutta de Ordem 2,** obtemos que para passo h=0,5 a velocidade do corpo v(t) no instante t=5 segundos será de **49,374923 m/s**

**b)** Usando o método de **Runge-Kutta de Ordem 4** para resolver o exercício, obtemos os seguintes resultados usando **h = 0.5**:

| **h = 0.5** | 0.000000 | 4.993750 | 9.975000 | 14.943750 | 19.900000 | 24.843750 | 29.775000 | 34.693750 | 39.600000 | 44.493750 | **49.375000** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Foram usados os valores de **x = 0** até **x = 5** para observar a aproximação de valores até obter o resultado final de **y(5)** sendo o valor de **49.375000.**

## **Exercício 8**

Partindo da equação , a qual é um exemplo de Equação de Bernoulli, podemos calcular usando um programa que soluciona a EDO apresentada. Desse modo, utilizando h = 5, apresentamos os resultados obtidos com o intuito de chegar a y(30). De acordo com o enunciado, **y(0) = 1000** e as constantes são **k = 0,000002** e **m = 1000000**, logo após **t = 30 dias**, o número de infectados cresce da maneira apresentada na tabela abaixo:

| **t** | **0** | **5** | **10** | **15** | **20** | **25** | **30** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **y(t)** | 1000.000000 | 2663.432581 | 6903.165620 | 16741.137992 | 35306.850744 | 59719.189280 | **80092.596001** |

Sendo assim, o número de infectados após 30 dias é **80092 infectados** aproximadamente**.**

## **Exercício 9**

Para o exercício 9 foi requisitado que os programas fossem feitos em código na linguagem em Python. Logo, os mesmos estão feitos no Anexo - Parte 3, respectivamente.

# **Parte 4**

## **Exercício 1**

A partir da equação + isolamos a taxa de variação da carga pelo tempo para adequar a EDO ao programa. Logo, a equação reescrita ficou , assim aplicando os valores de R = 2 , C = 3 F e V = 4 V, que são constantes no problema, a resolução para os resultados entre **[0,4]** com **h = 0.5** foram:

| **t** | **0.0** | **0.5** | **1.0** | **1.5** | **2.0** | **2.5** | **3.0** | **3.5** | **4.0** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Q(t)** | 0.000000 | 0.959467 | 1.842219 | 2.654390 | 3.401623 | 4.089111 | 4.721631 | 5.303577 | 5.838993 |

## **Exercício 2**

Para o exercício referente a população foi utilizado o **Método de Runge-Kutta de Ordem 4** para encontrar a solução **P(10)**. Usando os dados do enunciado e **h = 1.0**, os resultados apresentados de [0,10] são:

| **t** | **0.0** | **1.0** | **2.0** | **3.0** | **4.0** | **5.0** | **6.0** | **7.0** | **8.0** | **9.0** | **10.0** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P(t)** | 1000.000000 | 1000.143333 | 1001.106667 | 1003.690000 | 1008.693333 | 1016.916667 | 1029.160000 | 1046.223333 | 1068.906667 | 1098.01000 | **1134.33333** |

Foram apresentados os resultados no intervalo de **[0,10]** para observarmos a variação de valores conforme as iterações eram feitas. Assim, o resultado final de **P(10)** foi de **1134.33333.**

## **Exercício 3**

Para o método de Euler, considerou-se a seguinte equação:

Tn+1 = Tn + h \* f(tn, Tn)

T0 = T(t0) = 293 ⇒ condição inicial

t = tempo

Considerando o valor de h = 0.001, realizou-se o cálculo iterativo com base no valor inicial dado no problema e a função f, definida por:

f(T) =

Logo, a resolução do problema pelo método foi realizado da seguinte maneira:

T1 = 293 + 0.001\*(

T2 = + 0.001\*(

.

.

.

Para esta iteração, utilizou-se a programação em Python como ferramenta a fim de encontrar a solução pelo método.

Já para a solução exata, foi resolvida a EDO através do método do fator integrante, da seguinte maneira

Equação:

p(t) =

q(t) = 0,313

i(t) = =

T =

Resolvendo a equação, encontra-se a solução :

Comparando os valores, temos que para uma precisão de 95% (abs(T aprox. - T soluçãoExata)/T soluçãoExata), o tempo mínimo foi de 245.45 seg.

## 

## **Exercício 4**

A equação utilizada para o método de Euler é:

Utilizando o método de Euler

então

Supondo h=3.1 e n=6

1.75

1.53

1.33

1.13

1

Resolvendo a edo de solução exata:

m(t) = 1, m(t) = 15,5

Sendo assim, em 15,5 minutos é reduzido a massa pela metade e sendo comprovado pela solução exata.

## **Exercício 5**

Por meio do **Método de Euler** no formato:

, com **h= 0,05** e PVI **y(0)=10²**, substituindo na fórmula temos que yn+1=yn+(0,05\*0,1\*(yn\*(½))\*(1000-yn)/ √|1000-yn| , assim com o auxílio da programação em linguagem python foi possível solucionar o problema, constatando que a população de bactérias se satura no tempo de **24,85 minutos** e ela atinge 90% dessa quantidade em **18,55 minutos.**

# **Anexos - parte 1**

## **Exercício 1**

**itr = 30**

**x1 = -2**

**x2 = 5**

**h = (x2 - x1)/itr**

**res = ((x1\*\*4 - 3\*x1\*\*3+ 2\*x1\*\*2 - 3) + (x2\*\*4 - 3\*x2\*\*3+ 2\*x2\*\*2 - 3))/2**

**i=1**

**while i<itr:**

**x = -2 + (i\*h)**

**res += x\*\*4 - 3\*x\*\*3+ 2\*x\*\*2 - 3**

**i +=1**

**res = res \*h**

**print(res)**

## **Exercício 2**

**Regra dos trapézios**

**a)**

**def f(x):**

**return 2.71828182845904523536\*\*x**

**itr = 4**

**x1 = 1**

**x2 = 2**

**h = (x2 - x1)/itr**

**res = (f(x1) + f(x2))/2**

**i=1**

**while i <itr:**

**x = x1 + (i\*h)**

**res += (f(x))**

**i +=1**

**res = res \*h**

**print(res)**

**b)**

**def f(x):**

**return x\*\*(1/2)**

**itr = 4**

**x1 = 1**

**x2 = 4**

**h = (x2 - x1)/itr**

**res = (f(x1) + f(x2))/2**

**print(res)**

**i=1**

**while i <itr:**

**x = x1 + (i\*h)**

**res += (f(x))**

**i +=1**

**print(res)**

**res = res \*h**

**print(res)**

**c)**

**def f(x):**

**return 1/(x\*\*(1/2))**

**itr = 4**

**x1 = 2**

**x2 = 14**

**h = (x2 - x1)/itr**

**res = (f(x1) + f(x2))/2**

**i=1**

**while i <itr:**

**x = x1 + (i\*h)**

**res += (f(x))**

**i +=1**

**res = res \*h**

**print(res)**

**Regra de Simpson**

**import numpy as np**

**def funcao(n, x):**

**if(n == 1): #Exercicio 2 letra a**

**result = np.exp(x)**

**elif(n == 2): #Exercicio 2 letra b**

**result = x\*\*(0.5)**

**elif(n == 3): #Exercicio 2 letra c**

**result = x\*\*(-0.5)**

**elif(n == 4): #Exercicio 3 letra a**

**result = 1/(1+x)**

**else:**

**result = np.cos(x) #Exercicio 4**

**return result**

**def MetodoSimpson\_4Partes(xn, x0, n):**

**h = (xn - x0)/4**

**valor = funcao(n, x0) + 4\*funcao(n, x0+h) + 2\*funcao(n, x0+2\*h) + 4\*funcao(n, x0+3\*h) + funcao(n, xn)**

**return ((h/3)\*valor)**

**def MetodoSimpson\_6Partes(xn, x0, n):**

**h = (xn - x0)/6**

**valor = funcao(n, x0) + 4\*funcao(n, x0+h) + 2\*funcao(n, x0+2\*h) + 4\*funcao(n, x0+3\*h) + 2\*funcao(n, x0+4\*h) + 4\*funcao(n, x0+5\*h) + funcao(n, xn)**

**return ((h/3)\*valor)**

**print(MetodoSimpson\_4Partes(np.pi/2, 0, 5))**

**print(MetodoSimpson\_6Partes(np.pi/2, 0, 5))**

**Utilizando as funções implementadas acima, executar os seguintes comandos**

**a)**

**print(MetodoSimpson\_4Partes(2, 1, 1)) # ⇒ 4.670874883494676**

**print(MetodoSimpson\_6Partes(2, 1, 1)) # ⇒ 4.670794226633773**

**b)**

**print(MetodoSimpson\_4Partes(4, 1, 2)) # ==> 4.666220708306385**

**print(MetodoSimpson\_6Partes(4, 1, 2)) # ==> 4.6665630532224895**

**c)**

**print(MetodoSimpson\_4Partes(14, 2, 3)) # ==> 4.676374564596406**

**print(MetodoSimpson\_6Partes(14, 2, 3)) # ==> 4.66148949120848**

## **Exercício 4**

**a) Regra de Simpson**

**import numpy as np**

**def funcao(n, x):**

**if(n == 1): #Exercicio 2 letra a**

**result = np.exp(x)**

**elif(n == 2): #Exercicio 2 letra b**

**result = x\*\*(0.5)**

**elif(n == 3): #Exercicio 2 letra c**

**result = x\*\*(-0.5)**

**elif(n == 4): #Exercicio 3 letra a**

**result = 1/(1+x)**

**else:**

**result = np.cos(x) #Exercicio 4**

**return result**

**def MetodoSimpson\_4Partes(xn, x0, n):**

**h = (xn - x0)/4**

**valor = funcao(n, x0) + 4\*funcao(n, x0+h) + 2\*funcao(n, x0+2\*h) + 4\*funcao(n, x0+3\*h) + funcao(n, xn)**

**return ((h/3)\*valor)**

**def MetodoSimpson\_6Partes(xn, x0, n):**

**h = (xn - x0)/6**

**valor = funcao(n, x0) + 4\*funcao(n, x0+h) + 2\*funcao(n, x0+2\*h) + 4\*funcao(n, x0+3\*h) + 2\*funcao(n, x0+4\*h) + 4\*funcao(n, x0+5\*h) + funcao(n, xn)**

**return ((h/3)\*valor)**

**print(MetodoSimpson\_4Partes(0.6, 0, 4)) # ==> 0.47001715488409646**

**print(MetodoSimpson\_6Partes(0.6, 0, 4)) # ==> 0.47000638250638255**

**b) Regra dos trapézios**

**def f(x):**

**return 1/(1+x)**

**itr = 5**

**x1 = 0**

**x2 = 0.6**

**h = (x2 - x1)/itr**

**res = (f(x1) + f(x2))/2**

**i=1**

**while i <itr:**

**x = x1 + (i\*h)**

**res += (f(x))**

**i +=1**

**res = res \*h**

**print(res)**

## **Exercício 5**

**import numpy as np**

**def funcao(n, x):**

**if(n == 1): #Exercicio 2 letra a**

**result = np.exp(x)**

**elif(n == 2): #Exercicio 2 letra b**

**result = x\*\*(0.5)**

**elif(n == 3): #Exercicio 2 letra c**

**result = x\*\*(-0.5)**

**elif(n == 4): #Exercicio 3 letra a**

**result = 1/(1+x)**

**else:**

**result = np.cos(x) #Exercicio 4**

**return result**

**def MetodoSimpson\_4Partes(xn, x0, n):**

**h = (xn - x0)/4**

**valor = funcao(n, x0) + 4\*funcao(n, x0+h) + 2\*funcao(n, x0+2\*h) + 4\*funcao(n, x0+3\*h) + funcao(n, xn)**

**return ((h/3)\*valor)**

**def MetodoSimpson\_6Partes(xn, x0, n):**

**h = (xn - x0)/6**

**valor = funcao(n, x0) + 4\*funcao(n, x0+h) + 2\*funcao(n, x0+2\*h) + 4\*funcao(n, x0+3\*h) + 2\*funcao(n, x0+4\*h) + 4\*funcao(n, x0+5\*h) + funcao(n, xn)**

**return ((h/3)\*valor)**

**print(MetodoSimpson\_4Partes(np.pi/2, 0, 5)) # ==> 1.0001345849741938**

**print(MetodoSimpson\_6Partes(np.pi/2, 0, 5)) # ==> 1.0000263121705928**

# **Anexos - Parte 3**

## **Exercício 1**

**h = 0.01**

**n = 0**

**xn = 0**

**yn = 1**

**xfinal = 0.4**

**xn1 = 0**

**yn1 = 0**

**while xn <= xfinal:**

**print('Xn => {:.2f} Yn => {:.6f}'.format(xn, yn))**

**xn1 = xn + h**

**yn1 = yn + (0.01)\*(yn)**

**xn = xn1**

**yn = yn1**

**print('Xn => {:.2f} Yn => {:.6f}'.format(xn, yn))**

## **Exercício 2**

**h = 0.2**

**n = 0**

**xn = 0**

**yn = 0**

**xfinal = 1**

**xn1 = 0**

**yn1 = 0**

**while xn <= xfinal:**

**print('Xn => {:.2f} Yn => {:.12f}'.format(xn, yn))**

**xn1 = xn + h**

**yn1 = yn + (0.2)\*(4\*xn\*xn\*xn)**

**xn = xn1**

**yn = yn1**

## **Exercício 3**

Para h=0,1

h = 0.1

n = 0

xn = 2

yn = 2

xfinal = 2.1

xn1 = 0

yn1 = 0

while xn <= xfinal:

print('Xn => {:.2f} Yn => {:.12f}'.format(xn, yn))

xn1 = xn + h

yn1 = yn + (0.1)\*((xn - yn)/xn)

xn = xn1

yn = yn1

Para h=0,01

h = 0.01

n = 0

xn = 2

yn = 2

xfinal = 2.1

xn1 = 0

yn1 = 0

while xn <= xfinal:

print('Xn => {:.2f} Yn => {:.12f}'.format(xn, yn))

xn1 = xn + h

yn1 = yn + (0.01)\*((xn - yn)/xn)

xn = xn1

yn = yn1

Para h=0,025

h = 0.025

n = 0

xn = 2

yn = 2

xfinal = 2.1

xn1 = 0

yn1 = 0

while xn <= xfinal:

print('Xn => {:.2f} Yn => {:.12f}'.format(xn, yn))

xn1 = xn + h

yn1 = yn + (0.01)\*((xn - yn)/xn)

xn = xn1

yn = yn1

## **Exercício 4**

Para h= 0,5

**a)**

h = 0.5

n = 0

xn = 0

yn = 1000

xfinal = 1

xn1 = 0

yn1 = 0

while xn <= xfinal:

print('Xn => {:.2f} Yn => {:.12f}'.format(xn, yn))

xn1 = xn + h

yn1 = yn + (0.5)\*(0.04 \*yn)

xn = xn1

yn = yn1

Para h= 0,25

h = 0.25

n = 0

xn = 0

yn = 1000

xfinal = 1

xn1 = 0

yn1 = 0

while xn <= xfinal:

print('Xn => {:.2f} Yn => {:.12f}'.format(xn, yn))

xn1 = xn + h

yn1 = yn + (0.25)\*(0.04 \*yn)

xn = xn1

yn = yn1

Para h= 0,1

h = 0.1

n = 0

xn = 0

yn = 1000

xfinal = 1

xn1 = 0

yn1 = 0

while xn <= xfinal:

print('Xn => {:.2f} Yn => {:.12f}'.format(xn, yn))

xn1 = xn + h

yn1 = yn + (0.1)\*(0.04 \*yn)

xn = xn1

yn = yn1

**b)**

**Para h=0,5**

**def RungeKutta2(f,y,x,h):**

**k1 = h\*f(x,y)**

**k2 = lambda k1: h\*f(x + h/2, y + k1/2)**

**return y + k2(k1)**

**f = lambda x,y : 0.04\*y # Obtem se essa função isolando y' na EDO**

**y0,x0,h,t = 1000, 0, 0.5 ,1 # Parâmetros iniciais dados para a resolução do problema**

**# RUNGE-KUTTA 2a Ordem**

**print ('\n\n\t\t\t\tRunge-Kutta 2 Ordem\n\n')**

**while x0 <= t:**

**yn = RungeKutta2(f,y0,x0,h)**

**print("y(%2.2f)\t= %4.6f" % (x0,y0))**

**y0 = yn**

**x0 += h**

**Para h=0,25**

**def RungeKutta2(f,y,x,h):**

**k1 = h\*f(x,y)**

**k2 = lambda k1: h\*f(x + h/2, y + k1/2)**

**return y + k2(k1)**

**f = lambda x,y : 0.04\*y # Obtem se essa função isolando y' na EDO**

**y0,x0,h,t = 1000, 0, 0.25 ,1 # Parâmetros iniciais dados para a resolução do problema**

**# RUNGE-KUTTA 2a Ordem**

**print ('\n\n\t\t\t\tRunge-Kutta 2 Ordem\n\n')**

**while x0 <= t:**

**yn = RungeKutta2(f,y0,x0,h)**

**print("y(%2.2f)\t= %4.6f" % (x0,y0))**

**y0 = yn**

**x0 += h**

**Para h=0,1**

**def RungeKutta2(f,y,x,h):**

**k1 = h\*f(x,y)**

**k2 = lambda k1: h\*f(x + h/2, y + k1/2)**

**return y + k2(k1)**

**f = lambda x,y : 0.04\*y # Obtem se essa função isolando y' na EDO**

**y0,x0,h,t = 1000, 0, 0.1 ,1 # Parâmetros iniciais dados para a resolução do problema**

**# RUNGE-KUTTA 2a Ordem**

**print ('\n\n\t\t\t\tRunge-Kutta 2 Ordem\n\n')**

**while x0 <= t:**

**yn = RungeKutta2(f,y0,x0,h)**

**print("y(%2.2f)\t= %4.6f" % (x0,y0))**

**y0 = yn**

**x0 += h**

**c)**

from math import sqrt

from math import exp

def RungeKutta4(f,y,x,h):

k1 = h\*f(x,y)

k2 = lambda k1: h\*f(x + h/2, y + k1/2)

k3 = lambda k2: h\*f(x + h/2, y + k2(k1)/2)

k4 = lambda k3: h\*f(x+h,y + k3(k2))

return y + (k1+2\*k2(k1)+2\*k3(k2)+k4(k3))/6

f = lambda x,y : 0.04\*y # Obtem se essa função isolando y' na EDO

y0,x0,h,t = 1000, 0, 0.1 ,1 # Parâmetros iniciais dados para a resolução do problema

# RUNGE-KUTTA 4a Ordem

print ('\n\n\t\t\t\tRunge-Kutta 4 Ordem\n\n')

while x0 <= 1.0:

yn = RungeKutta4(f,y0,x0,h)

print("y(%2.2f)\t= %4.6f" % (x0,y0))

y0 = yn

x0 += h

## **Exercício 5**

from math import sqrt

from math import exp

def RungeKutta4(f,y,x,h):

k1 = h\*f(x,y)

k2 = lambda k1: h\*f(x + h/2, y + k1/2)

k3 = lambda k2: h\*f(x + h/2, y + k2(k1)/2)

k4 = lambda k3: h\*f(x+h,y + k3(k2))

return y + (k1+2\*k2(k1)+2\*k3(k2)+k4(k3))/6

f = lambda x,y : ((y\*(x+h)-y)/h) # Obtem se essa função isolando y' na EDO

y0,x0,h,t = 1.5, 1, 0.2 , 1.6 # Parâmetros iniciais dados para a resolução do problema

# RUNGE-KUTTA 4a Ordem

print ('\n\n\t\t\t\tRunge-Kutta 4 Ordem\n\n')

while x0 <= 1.7:

yn = RungeKutta4(f,y0,x0,h)

print("y(%2.2f)\t= %4.6f" % (x0,y0))

y0 = yn

x0 += h

## **Exercício 6**

1. Este exercício foi feito analiticamente, como pedido no enunciado, logo, não há código.

from math import sqrt

from math import exp

def RungeKutta4(f,y,x,h):

k1 = h\*f(x,y)

k2 = lambda k1: h\*f(x + h/2, y + k1/2)

k3 = lambda k2: h\*f(x + h/2, y + k2(k1)/2)

k4 = lambda k3: h\*f(x+h,y + k3(k2))

return y + (k1+2\*k2(k1)+2\*k3(k2)+k4(k3))/6

f = lambda x,y : -k\*y # Obtem se essa função isolando y' na EDO

k = 0.173286

y0,x0,h,t = 10, 0, 1.0 , 1 # Parâmetros iniciais dados para a resolução do problema

# RUNGE-KUTTA 4a Ordem

print ('\n\n\t\t\t\tRunge-Kutta 4 Ordem\n\n')

while x0 <= 10:

yn = RungeKutta4(f,y0,x0,h)

print("y(%2.2f)\t= %4.6f" % (x0,y0))

y0 = yn

x0 += h

## **Exercício 7**

**a)**

**def RungeKutta2(f,y,x,h):**

**k1 = h\*f(x,y)**

**k2 = lambda k1: h\*f(x + h/2, y + k1/2)**

**return y + k2(k1)**

**f = lambda x,y : (2000 - 2\*y)/(200 - x) # Obtem se essa função isolando y' na EDO**

**y0,x0,h,t = 0, 0, 0.5 ,5 # Parâmetros iniciais dados para a resolução do problema**

**# RUNGE-KUTTA 2a Ordem**

**print ('\n\n\t\t\t\tRunge-Kutta 2 Ordem\n\n')**

**while x0 <= t:**

**yn = RungeKutta2(f,y0,x0,h)**

**print("y(%2.2f)\t= %4.6f" % (x0,y0))**

**y0 = yn**

**x0 += h**

**b)**

from math import sqrt

from math import exp

def RungeKutta4(f,y,x,h):

k1 = h\*f(x,y)

k2 = lambda k1: h\*f(x + h/2, y + k1/2)

k3 = lambda k2: h\*f(x + h/2, y + k2(k1)/2)

k4 = lambda k3: h\*f(x+h,y + k3(k2))

return y + (k1+2\*k2(k1)+2\*k3(k2)+k4(k3))/6

f = lambda x,y : (2000 - 2\*y)/(200 - x) # Obtem se essa função isolando y' na EDO

y0,x0,h,t = 0, 0, 0.5 ,5 # Parâmetros iniciais dados para a resolução do problema

# RUNGE-KUTTA 4a Ordem

print ('\n\n\t\t\t\tRunge-Kutta 4 Ordem\n\n')

while x0 <= 5.0:

yn = RungeKutta4(f,y0,x0,h)

print("y(%2.2f)\t= %4.6f" % (x0,y0))

y0 = yn

x0 += h

## **Exercício 8**

from math import sqrt

from math import exp

def RungeKutta4(f,y,x,h):

k1 = h\*f(x,y)

k2 = lambda k1: h\*f(x + h/2, y + k1/2)

k3 = lambda k2: h\*f(x + h/2, y + k2(k1)/2)

k4 = lambda k3: h\*f(x+h,y + k3(k2))

return y + (k1+2\*k2(k1)+2\*k3(k2)+k4(k3))/6

f = lambda x,y : (k\*(m - y))\*y # Obtem se essa função isolando y' na EDO

m = 100000

k = 0.000002

y0,x0,h,t = 1000, 0, 5. , 30 # Parâmetros iniciais dados para a resolução do problema

# RUNGE-KUTTA 4a Ordem

print ('\n\n\t\t\t\tRunge-Kutta 4 Ordem\n\n')

while x0 <= 30.0:

yn = RungeKutta4(f,y0,x0,h)

print("y(%2.2f)\t= %4.6f" % (x0,y0))

y0 = yn

x0 += h

# **Anexos - Parte 4**

## **Exercício 1**

from math import sqrt

from math import exp

def RungeKutta4(f,y,x,h):

k1 = h\*f(x,y)

k2 = lambda k1: h\*f(x + h/2, y + k1/2)

k3 = lambda k2: h\*f(x + h/2, y + k2(k1)/2)

k4 = lambda k3: h\*f(x+h,y + k3(k2))

return y + (k1+2\*k2(k1)+2\*k3(k2)+k4(k3))/6

f = lambda x,y : (U/R) - (y/(R\*C))# Obtem se essa função isolando y' na EDO

U = 4

R = 2

C = 3

y0,x0,h,t = 0, 0, 0.5 ,5 # Parâmetros iniciais dados para a resolução do problema

# RUNGE-KUTTA 4a Ordem

print ('\n\n\t\t\t\tRunge-Kutta 4 Ordem\n\n')

while x0 <= 4.0:

yn = RungeKutta4(f,y0,x0,h)

print("y(%2.2f)\t= %4.6f" % (x0,y0))

y0 = yn

x0 += h

## **Exercício 2**

from math import sqrt

from math import exp

def RungeKutta4(f,y,x,h):

k1 = h\*f(x,y)

k2 = lambda k1: h\*f(x + h/2, y + k1/2)

k3 = lambda k2: h\*f(x + h/2, y + k2(k1)/2)

k4 = lambda k3: h\*f(x+h,y + k3(k2))

return y + (k1+2\*k2(k1)+2\*k3(k2)+k4(k3))/6

f = lambda x,y : (a + (b\*x))\*x # Obtem se essa função isolando y' na EDO

a = 0.02

b = 0.4

y0,x0,h,t = 1000, 0, 1. , 10 # Parâmetros iniciais dados para a resolução do problema

# RUNGE-KUTTA 4a Ordem

print ('\n\n\t\t\t\tRunge-Kutta 4 Ordem\n\n')

while x0 <= 10.0:

yn = RungeKutta4(f,y0,x0,h)

print("y(%2.2f)\t= %4.6f" % (x0,y0))

y0 = yn

x0 += h

## **Exercício 3**

import numpy as np

def f(t,u):

return 10\*\*(-3)\*(313-u)

#tamanho e num. de passos

h = 0.001

#C.I.

t0 = 0

u0 = 293

temp = 0

#iteracoes

while(True):

t = t0 + h

u = u0 + h\*f(t0,u0)

T = 313 - 20\*np.exp(-(10\*\*(-3))\*temp)

if((abs(u0 - T)/T) >= 0.05):

break

u0 = u

t0 = t

temp+=0.03

#imprime

print("%1.4f" % (u))

print(str(T))

print((abs(u0 - T)/T))

## **Exercício 4**

**import numpy as np**

**#define f(t,u)**

**def f(t,u):**

**return -0.05 \* (u)\*\*(2/3)**

**#tamanho e num. de passos**

**h = 3.1**

**N = 6**

**#cria vetor t e u**

**t = np.empty(N)**

**u = np.copy(t)**

**#C.I.**

**t[0] = 0**

**u[0] = 2**

**#iteracoes**

**for i in np.arange(N-1):**

**t[i+1] = t[i] + h**

**u[i+1] = u[i] + h\*f(t[i],u[i])**

**#imprime**

**for i,tt in enumerate(t):**

**print("%1.1f %1.4f" % (t[i],u[i]))**

## **Exercício 5**

**alfa = 0.1**

**L = 1000**

**h = 0.05**

**xn1 = 0**

**yn1 = 0**

**xn = 0**

**yn = 100**

**stop = 0**

**populacaoSaturada = 0**

**while stop >= 0:**

**print('Xn => {:.2f} Yn => {:.6f}'.format(xn, yn))**

**xn1 = xn + h**

**yn1 = yn + (h\*alfa\*(yn \*\* (1/2))\*((L - yn)/(abs((L - yn) \*\* (1/2)))))**

**tempoSaturada = xn**

**xn = xn1**

**stop = yn1 - yn**

**populacaoSaturada = yn1**

**yn = yn1**

**print('populacaoSaturada no tempo = {:.6f}'.format(tempoSaturada))**

**populacaoSaturada = populacaoSaturada \* 0.9**

**h = 0.05**

**xn1 = 0**

**yn1 = 0**

**xn = 0**

**yn = 100**

**while yn <= populacaoSaturada:**

**xn1 = xn + h**

**yn1 = yn + (h\*alfa\*(yn \*\* (1/2))\*((L - yn)/(abs((L - yn) \*\* (1/2)))))**

**xn = xn1**

**yn = yn1**

**print('resultado : Xn => {:.2f} Yn => {:.6f}'.format(xn, yn))**