WESLEYGON?ALVESDASILVA_11233140_AERON?UTICA

October 27, 2020

1 Questão 1 (a)

A fim de resolver o problema por meio do método da bisseção, é necessário que seja obtido, apartir das curvas, uma terceira função, sendo as raizes dessa x, as coordenadas no eixo horizontal onde as duas funções se interceptam.

Igualando ambas funções:

$$-x^4 + 7.7x^3 - 18x^2 + 13.6x = -x^2 + 5x + 0.75 \tag{1}$$

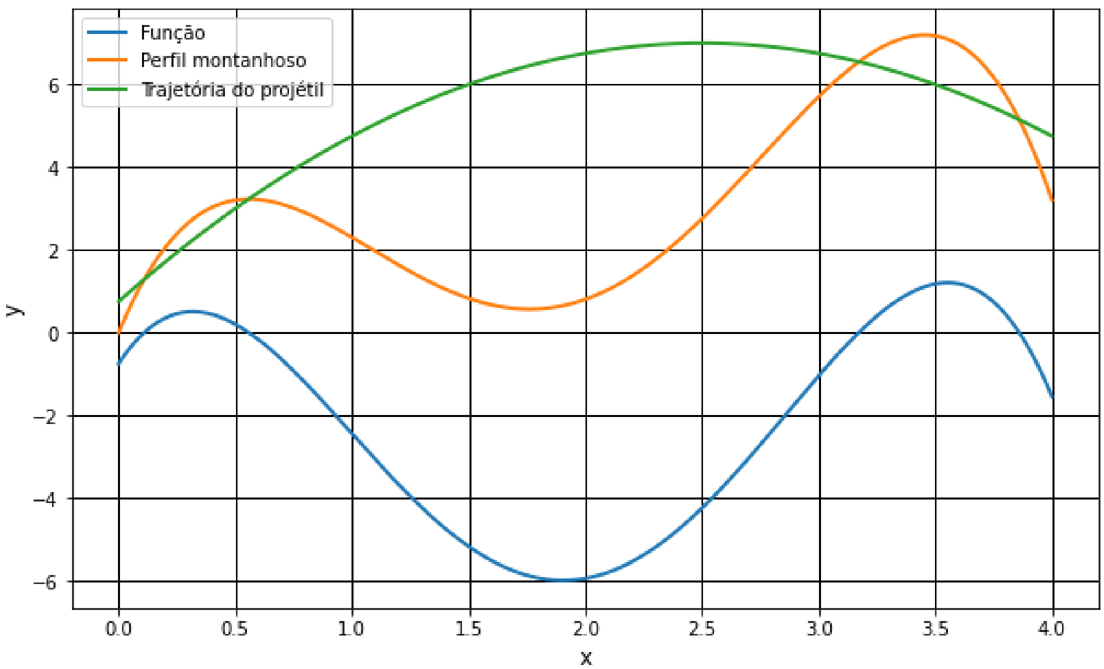
Obtendo-se:

$$-x^4 + 7.7x^3 - 17x^2 + 8.6x - 0.75 = 0 \tag{2}$$

Recorrendo-se a uma ferramente de plot.

Método da bisseção

[2]:
```
import numpy as np
import math
import matplotlib.pyplot as plt


func = lambda x: -x**4 + 7.7*x**3 - 17*x**2 + 8.6*x - 0.75
func1 = lambda x: -x**4 + 7.7*x**3-18*x**2+13.6*x
func2 = lambda x: -x**2+5*x+0.75
xi = np.linspace(0, 4, num=1001, endpoint=True)
yi = func(xi)
yi1 = func1(xi)
yi2 = func2(xi)
plt.figure(figsize=(10,6),facecolor='white')
plt.grid(color='k', linestyle='-', linewidth=1)
plt.plot(xi,yi,label = 'Função',linewidth = 2)
plt.plot(xi,yi1,label = 'Perfil montanhoso',linewidth = 2)
plt.plot(xi,yi2,label = 'Trajetória do projétil',linewidth = 2)
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('')
plt.legend()
plt.show()
```



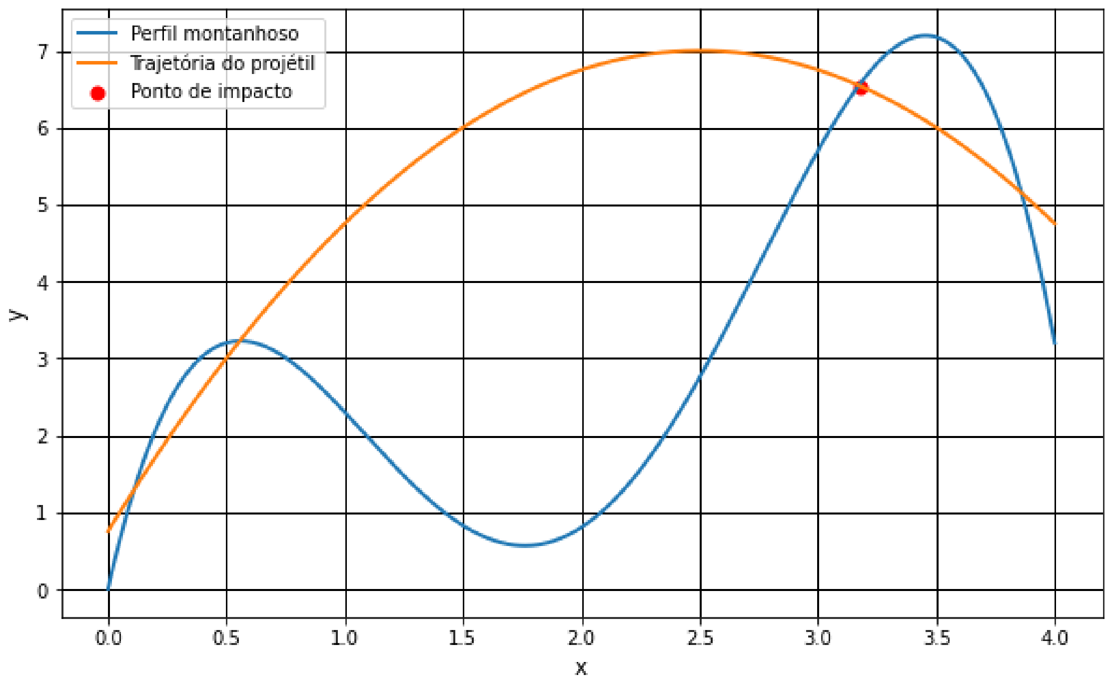
Naturalmente, como há 4 pontos de intersecção entre as funções fornecidas, o polinômio resultante possui 4 raízes. Como se quer obter a altura máxima a qual o impacto ocorre, a raíz de interesse está no intervalo $3 \leq x \leq 3.5$.

Portanto, pelo método da bisseção.

[5]:
```python
#Importe de bibliotecas

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

#Definição de parâmetros

func = lambda x: -x**4 + 7.7*x**3 - 17*x**2 + 8.6*x - 0.75 #Definição das curvas
func1 = lambda x: -x**4 + 7.7*x**3-18*x**2+13.6*x
func2 = lambda x: -x**2+5*x+0.75                            #Definição das curvas
a = 3                                                       #Intervalo de análise
b = 3.5
tol = 10**(-3)                                      #tolerância segundo enunciado
kmax = 5                          #Número máximo de iterações segundo enunciado
```

```python
def bissecaok(func,a,b,tol,kmax):                       #Definição do método da␣
↪bisseção
    x0 = (a+b)/2
    k = 0
    x = x0; erro = np.inf;
    while(erro > tol and k < kmax):
        k = k + 1;
        if(func(a)*func(x) < 0):
          b = x;
        else:
          a = x;
        x0 = x;
        x = (a+b)/2;
        erro = abs(x-x0);
    y = func2(x)                                        #Coordenada y de impacto

    print('Coordenadas do ponto de impacto: (%.6f,%.6f)' %(x,y)) #Impressão de␣
↪coordenadas
    print('\n')

    xi = np.linspace(0, 4, num=1001, endpoint=True)                #Plot
    yi1 = func1(xi)
    yi2 = func2(xi)
    plt.figure(figsize=(10,6),facecolor='white')
    plt.grid(color='k', linestyle='-', linewidth=1)
    plt.plot(xi,yi1,label = 'Perfil montanhoso',linewidth = 2)
    plt.plot(xi,yi2,label = 'Trajetória do projétil',linewidth = 2)
    plt.scatter(x,y,label = 'Ponto de impacto',linewidth = 2, color = 'r')
    plt.xlabel('x',fontsize='large')
    plt.ylabel('y',fontsize='large')
    plt.title('')
    plt.legend()
    plt.show()
bissecaok(func,a,b,tol,kmax)
```

```
Coordenadas do ponto de impacto: (3.179688,6.538025)
```



2 Questão 1 (b)

Por sua vez, apartir das funções da trajetória e perfil montanhoso, pode-se definir uma função de R^2 de forma que sua imagem também esteja em R^2 como:

$$F(x,y) = (f_1(x,y), f_2(x,y))^T \tag{3}$$

Onde as funções $f_1(x,y)$ e $f_2(x,y)$ são definidas como:

$$f_1(x,y) = -x^4 + 7.7x^3 - 18x^2 + 13.6x - y = 0 \tag{4}$$

e, também:

$$f_1(x,y) = -x^2 + 5x + 0,75 - y = 0 \tag{5}$$

O que permite aplicar o método de newtom para sistemas lineares para o seguinte sistema não linear.

$$\begin{cases} -x^4 + 7.7x^3 - 18x^2 + 13.6x - y = 0 \\ -x^2 + 5x + 0,75 - y = 0 \end{cases} \tag{6}$$

e, consequentemente:

$$F(x,y) = (f_1(x,y), f_2(x,y))^T = (0,0)^T \tag{7}$$

Assim, aplicando-se o algoritmo.

[34]:
```python
import numpy as np              #import de bibliotecas
import math

# Definindo funções e matriz do sistema

f1 = lambda x: -1*x[0]**4 + 7.7*x[0]**3 - 18*x[0]**2 + 13.6*x[0] - x[1]
f2 = lambda x: -1*x[0]**2 + 5*x[0] + 0.75 - x[1]
F = lambda x: np.array([f1(x),f2(x)])

# Definindo a função matricial Jacobiana

jac11 = lambda x: -4*x[0]**3 + 23.1*x[0]**2 - 36*x[0] +13.6
jac12 = lambda x: -1;
jac21 = lambda x: -2*x[0] + 5;
jac22 = lambda x: -1;
Jac = lambda x: np.array([[jac11(x),jac12(x)],[jac21(x),jac22(x)]])

x0 = [3,6];                     # chute inicial, próximo do ponto máximo

def newton_sis(F,Jac,x,tol,kmax):   #Definindo o comando de resolução por␣
 ↪Newton
    erro = np.inf; k = 0;
    while(erro > tol and k < kmax):
        k = k+1;
        v = np.linalg.solve(Jac(x),F(x));
        x = x-v;
        erro = np.linalg.norm(v);
    print('Coordenadas do ponto de impacto: (%.3f,%.3f)' %(x[0],x[1]))
    return x,k

(x,k) = newton_sis(F,Jac,x0,10**-3,1000)    #Aplicação do comando
```

```
Coordenadas do ponto de impacto: (3.173,6.547)
```

Observa-se que ambas soluções não são exatamente iguais justamente porque as funções apresentam critérios de parada diferentes, sendo o do primeiro algoritmo diferença absoluta e do segundo a norma do vetor resíduo.

3 Questão 2 (a)

As interpolações polinomiais consistem em, através de um conjunto de $n+1$ dados $A = \{(x_0, y_0), (x_2, y_2), ..., (x_i, y_i), ..., (x_n, y_n)\}$, obter um polinômio $P_n(x)$ (chamado de polinômio interpolante) de grau n, de tal forma que:

$$P_n(x_i) = y_i \tag{8}$$

Com o polinomio obtido através de diferentes métodos, neste caso através do método de Lagrange e pelo método de Newton, se obtém uma função contínua que contempla os pontos fornecidos, sendo assim, uma poderosa estratégia para descrever aproximações de curvas tendo como base dados pontuais da mesma, sendo esses chamados de nós da interpolação.

Para um dado conjunto de dados, pelo teorema visto em aula, o seu polinômio interpolante é único, o que é verificado facilmente, visto que para cada x_i, há apenas uma correspondência y_i, de forma que:

$$a_0 + a_1 x_i + a_2 x_i^2 + a_3 x_i^3 + ... + a_n x_i^n = y_i \tag{9}$$

Sendo os coeficientes a_k os únicos parâmetros a serem determinados. Portanto, para cada $(n+1)$ dado, tem-se uma linha de um sistema linear, cuja solução é única, visto que o determinante da matriz do sistema (que é da forma da matriz de Vandermonde) é dado por:

$$det(\mathbf{A}) = \prod_{i<k}^{n} (x_k - x_i) \neq 0 \tag{10}$$

$$\begin{pmatrix} 1 & x_0 & x_0^2 & ... & x_0^n \\ 1 & x_1 & x_1^2 & ... & x_1^n \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_n & x_n^2 & ... & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} \tag{11}$$

Visto que $x_k \neq x_i$.

Os métodos de Newton e Lagrange têm como fim simplificar o cálculo dos coeficientes do polinômio interpolador, de modo a evitar a solução de um sistema com $n+1$ incógnitas.

O método de Lagrange consiste em obter um polinômio da seguinte forma:

$$P_n(x) = y_0 l_0(x) + y_1 l_1(x) + ... + y_n l_n(x) \tag{12}$$

Onde os termos $l_k(x)$ são obtidos da seguinte forma:

$$l_k(x) = \prod_{i<k, i\neq k}^{n} \frac{x - x_i}{x_k - x_i} \tag{13}$$

Para $k = 0, 1, 2, ...n$

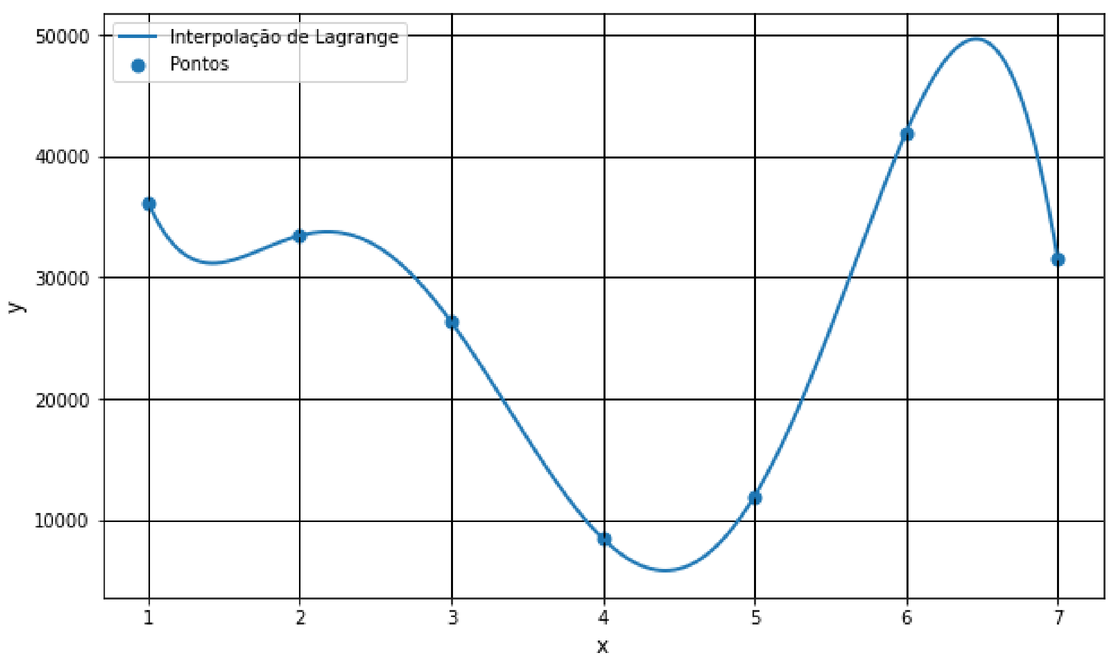
Os termos $l_k(x)$ (que são polinômios), tem como suas raízes todo o x_i onde $x_i \neq x_k$ e, assume valor 1 para $x_i = x_k$, ou seja, o polinômio contempla os pontos fonecidos, visto que seus termos são nulos para qualquer x_i que esteja no conjunto de dados, com exceção de um deles, que por sua vez assumirá valor 1 e, por estar multiplicado por y_i, par correspondente de x_i em questão, contempla o par (x_i, y_i).

Abaixo, tem-se o algoritmo para a interpolação pelo método de Lagrange dos pontos fornecidos pelo enunciado.

[12]:
```
import numpy as np                    #import de bibliotecas
import matplotlib.pyplot as plt

def lagrange_interp(xi,yi,x): #Definição da interpolação por lagrange
    n = np.size(xi);          # maior dimensão de um vetor unidimensional
    m = np.size(x);           # maior dimensão de um vetor unidimensional
    L = np.ones((n,m));       # matriz L
    for i in np.arange(n):
        for j in np.arange(n):
            if(i != j):
```

```
                L[i,:] = (L[i,:]*(x-xi[j]))/(xi[i]-xi[j]); #termo geral dos␣
↪elementos de L
    y = yi.dot(L); # --> yi é vetor linha 1xn L é matriz nxm --> 1xm
    return y;
```

[36]:
```
import numpy as np                #Import de bibliotecas
import matplotlib.pyplot as plt

#Dados fornecidos

xi = np.array([1,2,3,4,5,6,7], dtype='double');
yi = np.array([36175,33431,26310,8456,11946,41916,31553], dtype='double');

x = np.linspace(1, 7, num=1001, endpoint=True)

y_la = lagrange_interp(xi,yi,x)

#plot de solução

plt.figure(figsize=(10,6),facecolor='white')
plt.grid(color='k', linestyle='-', linewidth=1)
plt.plot(x,y_la,label = 'Interpolação de Lagrange',linewidth = 2)
plt.scatter(xi,yi,label = 'Pontos',linewidth = 2)
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('')
plt.legend()
plt.show()
```



Por sua vez, o método de Newton se baseia na seguinte expressão para o polinômio interpolador.

$$P_n(x) = \alpha_0 + \alpha_1(x - x_0) + \alpha_2(x - x_0)(x - x_1) + ... + \alpha_n(x - x_0)(x - x_1)...(x - x_{n-1}) \tag{14}$$

Onde os parâmetros do polinômio devem satisfazer o seguinte sistema:

$$\begin{cases} \alpha_0 = y_0 \\ \alpha_0 + \alpha_1(x_1 - x_0) = y_1 \\ \alpha_0 + \alpha_1(x_2 - x_0) + \alpha_2(x_2 - x_0)(x_2 - x_1) = y_2 \\ \downarrow \\ \alpha_0 + \alpha_1(x_n - x_0) + ... + \alpha_n(x_n - x_0)...(x_n - x_{n-1}) = y_n \end{cases} \tag{15}$$

Observa-se que cada linha do sistema, não utiliza dados da antecessora apenas termos do conjunto de dados fornecido, ou seja, não depende de sua antecessora e, portanto, é possível incluir dados adicionais para a interpolação sem que seja necessário recalcular-se todos os coeficientes αs anteriores.

De forma a calcular esses coeficientes de forma mais prática, emprega-se o conceito de diferença dividida e, portanto, o termo geral é:

$$\alpha_k = f[x_0, x_1, ..., x_k] \tag{16}$$

Para $k = \{0, 1, 2, ..., n\}$

Onde a mesma é calculada por:

$$f[x_i, x_{i+1}, ..., x_{i+k}] = \frac{f[x_{i+1}, x_{i+2}, ..., x_{i+k}] - f[x_i, x_{i+1}, ..., x_{i+k-1}]}{x_{i+k} - x_i} \tag{17}$$

Onde $k = \{0, 1, 2, ..., n\}$ e $i = \{0, 1, 2, ..., n - k\}$.

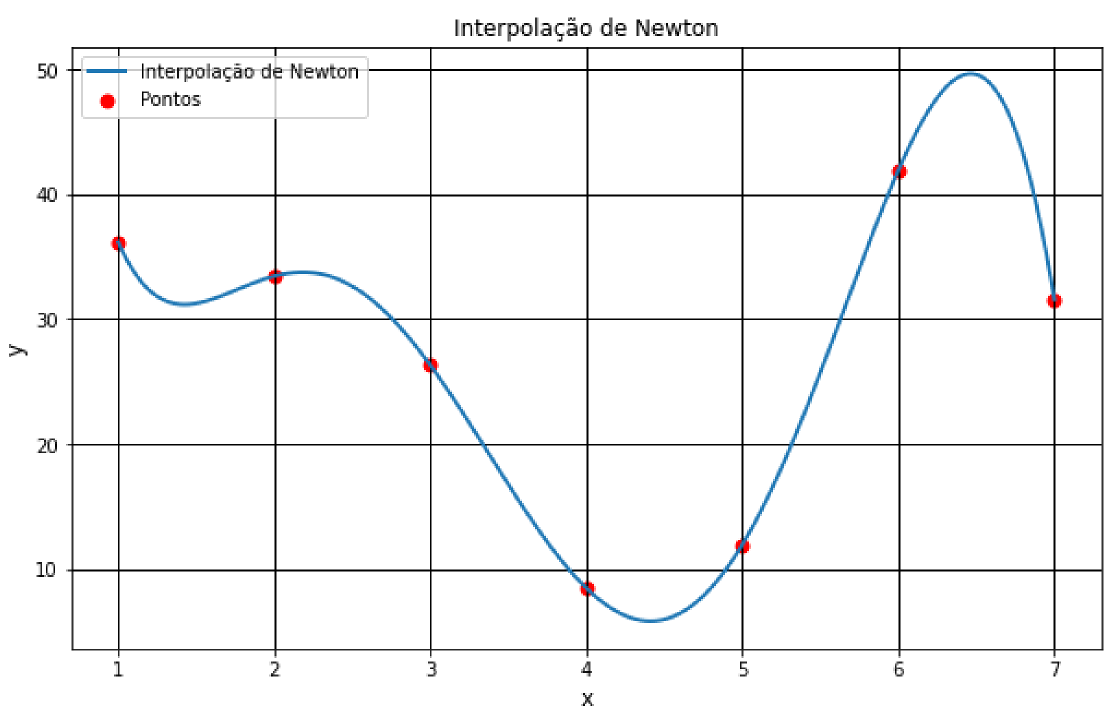
De forma que, para os temos iniciais do processo, tem-se simplesmente:

$$f[x_i] = y_i \tag{18}$$

Observa-se que este é um processo recursivo, de modo que, os termos iniciais do processo são simplesmente os termos correspondentes y_i e, portanto, pode-se calcular os termos sequentes.

[6]:
```python
def newton_interp(xi,yi,x):     #Definição da função interpolação
    n = np.size(xi); ni = np.size(x); N = np.ones((n,ni));
    D = np.zeros((n,n)); D[:,0] = yi;

    for j in np.arange(n-1): # matriz de diferenças divididas
        for i in np.arange(n-j-1):
            D[i,j+1] = (D[i+1,j]-D[i,j])/(xi[i+j+1]-xi[i]); #termo geral de D

    for i in np.arange(1,n): # loop do produtório da forma de Newton
        N[i,:] = N[i-1,:]*(x-xi[i-1]); #Produtório da forma de Newton

    y = D[0,:].dot(N)       # Imagem da curva para o intervalo x
    return y;
```

[8]:
```
import numpy as np                    #Import de bibliotecas
import matplotlib.pyplot as plt

#Dados

xi = np.array([1,2,3,4,5,6,7], dtype='double');
yi = np.array([36.175,33.431,26.310,8.456,11.946,41.916,31.553],␣
 ↪dtype='double');
x = np.linspace(1, 7, num=1001, endpoint=True)

#Chamado função de interpolação por Newton

y_ne = newton_interp(xi,yi,x)

#Plot da solução

plt.figure(figsize=(10,6),facecolor='white')
plt.grid(color='k', linestyle='-', linewidth=1)
plt.plot(x,y_ne,label = 'Interpolação de Newton',linewidth = 2)
plt.scatter(xi,yi,label = 'Pontos',linewidth = 2, facecolor='red')
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('Interpolação de Newton')
plt.legend()
plt.show()
```



A fim de se obter a expressão do polinômio interpolante numericamente, emprega-se a resolução do sistema definido pela matriz de Vandermonde (A), matriz de coeficientes do polinômio $(a_0, a_1, a_2, ..., a_n)$ e pela matriz (b) que contém os dados da imagem y_i. Resolvendo através do algoritmo abaixo, tem-se:

[9]:
```
def vandermonde(xi,yi):        #Definição da função
    n = np.size(xi)
    A = np.ones((n,n), dtype=np.float64)
    for i in range(0,n,1):
        for j in range(0,n,1):
            A[i,j] = xi[i]**(j)   #construção da matriz de coeficientes

    x = np.linalg.solve(A,yi)     #Resolução do sistema linear
    print(x)
```

[10]:
```
import numpy as np
from scipy import linalg as lin
xi = np.array([1,2,3,4,5,6,7], dtype='double');
yi = np.array([36175,33431,26310,8456,11946,41916,31553], dtype='double');

vandermonde(xi,yi)
```

```
[ 1.65071000e+05 -2.87067250e+05  2.28728519e+05 -8.41699583e+04
  1.47231944e+04 -1.13879167e+03  2.82861111e+01]
```

Portanto, ordenando os coeficientes obtidos tem-se o polinômio interpolador como:

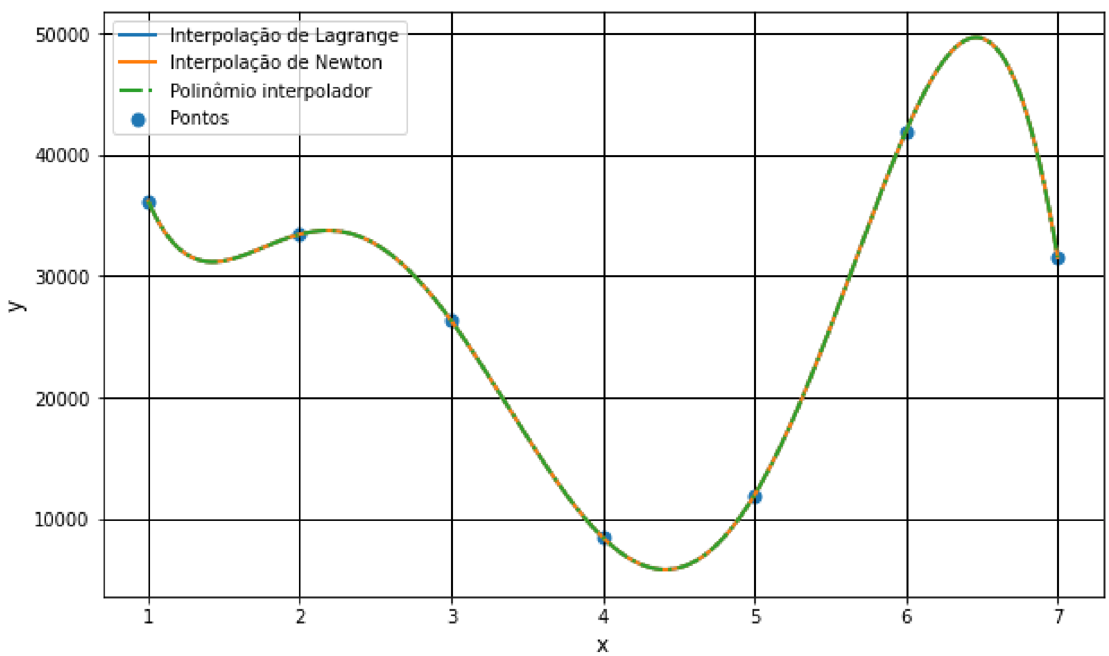
$$165071 - 287067.25x + 228728.519x^2 - 84169.958x^3 + 14723.194x^4 - 1138.791x^5 + 28.286x^6 \quad (19)$$

Onde que por questões estéticas, se representa o polinômio com 4 algarismos significativos.

Observa-se que as três curvas obtidas coincidem umas com as outras resultado em apenas uma curva visível.

[27]:
```
import numpy as np                #Import de biblitecas
import matplotlib.pyplot as plt

#Definição do polinômio

func = lambda x: 165071 - ((287067.25)*x) + ((228728.519)*x**(2)) - ((84169.
 ↪958)*x**(3)) + ((14723.194)*x**(4)) - ((1138.791)*x**(5)) + ((28.286)*x**6)

#Dados

xi = np.array([1,2,3,4,5,6,7], dtype='double');
yi = np.array([36175,33431,26310,8456,11946,41916,31553], dtype='double');

x = np.linspace(1, 7, num=1001, endpoint=True)
```

```
#Chamada de funções interpolação

y_la = lagrange_interp(xi,yi,x)
y_ne = newton_interp(xi,yi,x)

#Plot de polinômios

plt.figure(figsize=(10,6),facecolor='white')
plt.grid(color='k', linestyle='-', linewidth=1)
plt.plot(x,y_la,label = 'Interpolação de Lagrange', linewidth = 2)
plt.plot(x,y_ne,label = 'Interpolação de Newton', linestyle='-', linewidth = 2)
plt.plot(x,func(x),label = 'Polinômio interpolador', linestyle='-.', linewidth␣
 ↪= 2)
plt.scatter(xi,yi,label = 'Pontos',linewidth = 2)
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('')
plt.legend()
plt.show()
```

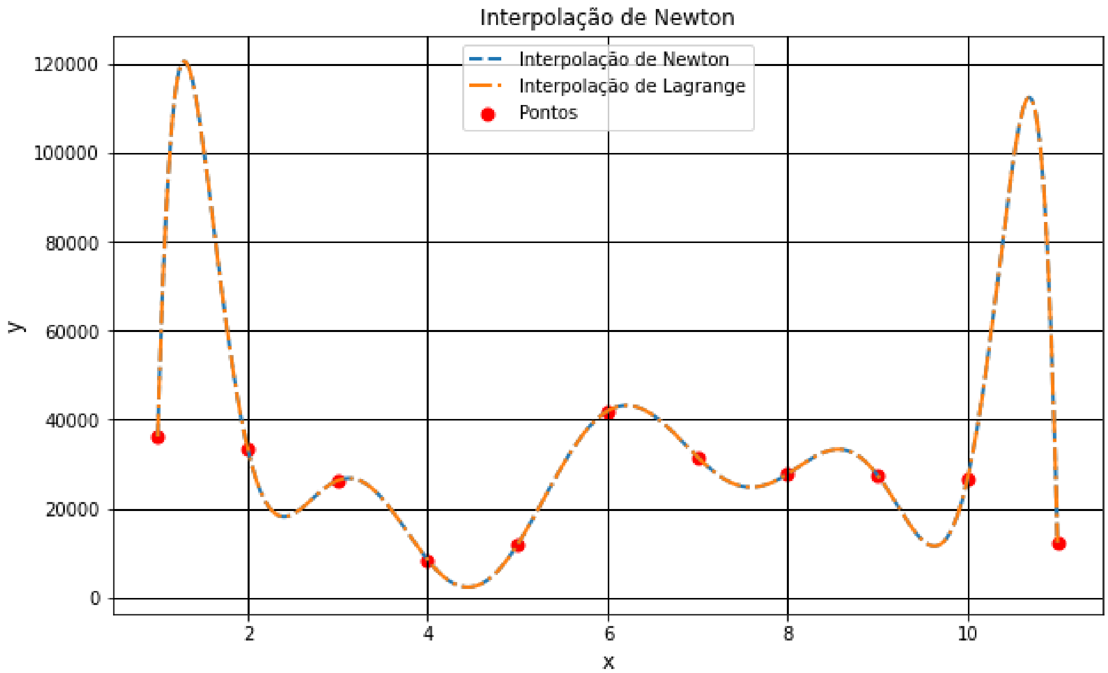


4 Questão 2 (b)

Com a adição de dados de mais 4 dias, tem-se as seguintes representações do polinômio interpolador.

[26]:

```
import numpy as np      #Import de bibliotecas
from scipy import linalg as lin

#Dados

xi1 = np.array([1,2,3,4,5,6,7,8,9,10,11], dtype='double');
yi1 = np.
 ↪array([36175,33431,26310,8456,11946,41916,31553,27750,27444,26749,12342],␣
 ↪dtype='double');

x1 = np.linspace(1, 11, num=1001, endpoint=True)

#Chamada de funções de interpolação

y_la1 = lagrange_interp(xi1,yi1,x1)
y_ne1 = newton_interp(xi1,yi1,x1)

#Plot do polinômio

plt.figure(figsize=(10,6),facecolor='white')
plt.grid(color='k', linestyle='-', linewidth=1)
plt.plot(x1,y_ne1,label = 'Interpolação de Newton', linestyle='--',linewidth =␣
 ↪2)
plt.plot(x1,y_la1,label = 'Interpolação de Lagrange', linestyle='-.', linewidth␣
 ↪= 2)
plt.scatter(xi1,yi1,label = 'Pontos',linewidth = 2, facecolor='red')
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('Interpolação de Newton')
plt.legend()
plt.show()
```



De modo que a espressão do polinômio interpolador obtida numéricamente é

[43]:
```
import numpy as np                  #Import de bibliotecas
from scipy import linalg as lin

#Dados fornecidos

xi1 = np.array([1,2,3,4,5,6,7,8,9,10,11], dtype='double');
yi1 = np.
 ↪array([36175,33431,26310,8456,11946,41916,31553,27750,27444,26749,12342],␣
 ↪dtype='double');

#Chamada da função

vandermonde(xi1,yi1)
```

```
[-6.67498700e+06  1.93164892e+07 -2.26109174e+07  1.44251834e+07
 -5.62023766e+06  1.40937664e+06 -2.32037390e+05  2.49221944e+04
 -1.68046814e+03  6.45433794e+01 -1.07669615e+00]
```

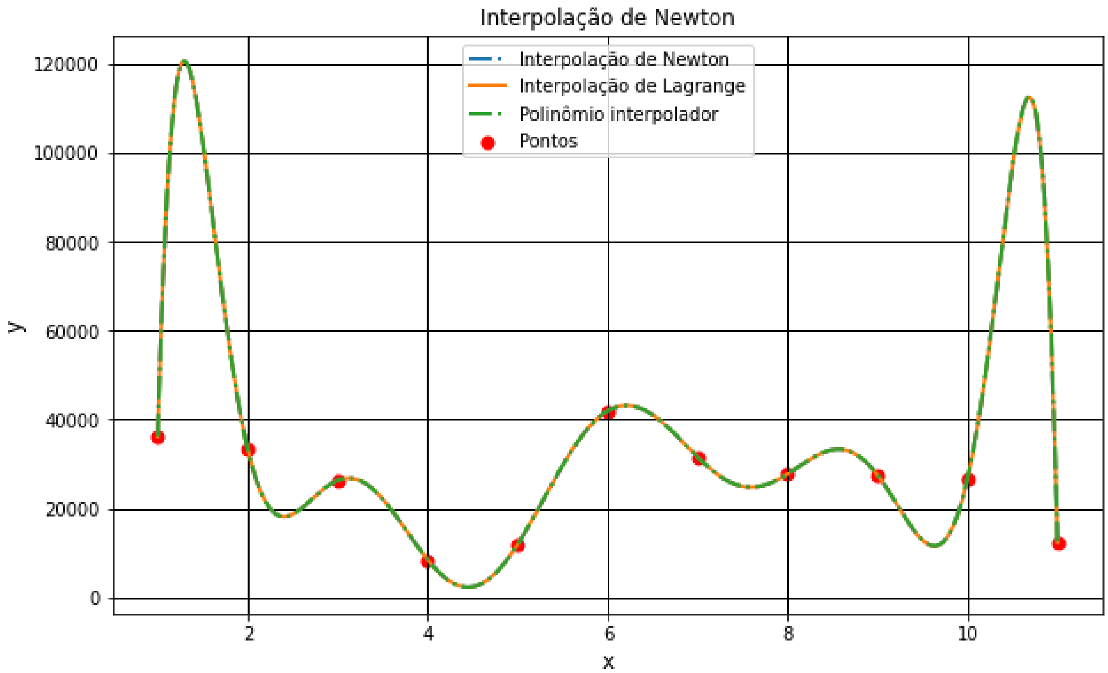
Portanto, tem-se como polinômio interpolador:

$$-6674987 + 19316489.2x - 22610917.4x^2 + 14425183.4x^3 - 5620237.66x^4 + 1409376.64x^5 - 232037.39x^6 + 24922.19 \quad (20)$$

Comparando os resultados obtidos, tem-se.

[31]:

```
import numpy as np        #Import de bibliotecas
import matplotlib.pyplot as plt

# Dados fornecidos

xi1 = np.array([1,2,3,4,5,6,7,8,9,10,11], dtype='double');
yi1 = np.
 ↪array([36175,33431,26310,8456,11946,41916,31553,27750,27444,26749,12342],␣
 ↪dtype='double');

x1 = np.linspace(1, 11, num=1001, endpoint=True)

#Polinômio interpolador

func1 = lambda x: - 6.67498700e+06 + (1.93164892e+07*x) - (2.
 ↪26109174e+07*(x**2)) + (1.44251834e+07*(x**3)) - (5.62023766e+06*(x**4)) +␣
 ↪(1.40937664e+06*(x**5)) - (2.32037390e+05*(x**6)) + (2.49221944e+04*(x**7))␣
 ↪- (1.68046814e+03*(x**8)) + (6.45433794e+01*(x**9)) - (1.
 ↪07669615e+00*(x**10))

#Chamada de funções

y_la1 = lagrange_interp(xi1,yi1,x1)
y_ne1 = newton_interp(xi1,yi1,x1)

#Plot de soluções

plt.figure(figsize=(10,6),facecolor='white')
plt.grid(color='k', linestyle='-', linewidth=1)
plt.plot(x1,y_ne1,label = 'Interpolação de Newton',linestyle='-.', linewidth =␣
 ↪2)
plt.plot(x1,y_la1,label = 'Interpolação de Lagrange', linestyle='-',linewidth =␣
 ↪2)
plt.plot(x1,func1(x1),label = 'Polinômio interpolador',linestyle='-.',linewidth␣
 ↪= 2)
plt.scatter(xi1,yi1,label = 'Pontos',linewidth = 2, facecolor='red')
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('Interpolação de Newton')
plt.legend()
plt.show()
```



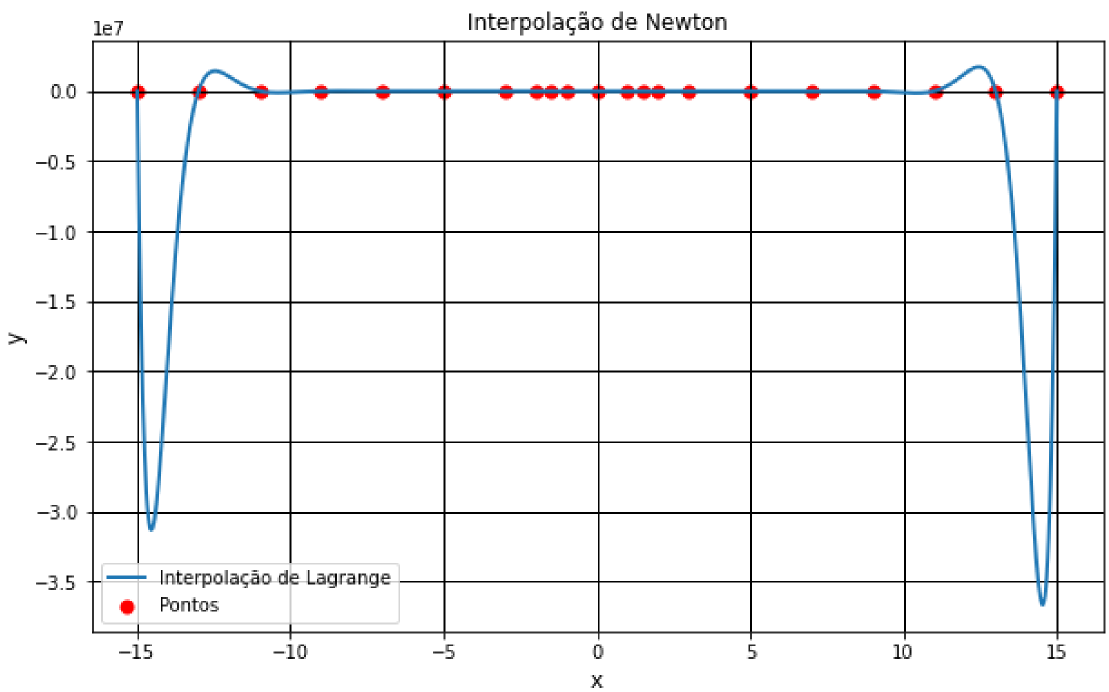
Observa-se, contudo que utilizar-se do método de Newton para contextos onde novos dados são possiveis de serem adicionados é mais vantajoso do que usar o método de Lagrange, justamente porque o segundo utiliza-se de um processo (produtório) que utiliza de todo o conjunto (com exceção do dado relativo ao próprio termo k) e, portanto, a adição de mais pontos torna todo o processo anterior inválido e, assim, é necessário fazer todos os cálculos dos termos $l_k(x)$ novamente. Contudo, tal dependencia não ocorre sobre o método de Newton, visto que, pelo sistema contido na seção sobre o método de Newton, a adição de mais dados não altera o cálculo de αs anteriores, resultando apenas na adição de mais uma linha do sistema, um α e seu polinômio multiplicativo.

5 Questão 3 (a)

Com base nos dados fornecidos, emprega-se o algoritmo sobre o método de Lagrange.

[32]:
```
import numpy as np      #Import de bibliotecas
import matplotlib.pyplot as plt

# Dados fornecidos

xi2 = np.array([-15,-13,-11,-9,-7,-5,-3,-2,-1.5,-1, 0,1.5, 1, 2, 3, 5, 7,␣
 ↪9,11,13,15], dtype='double');
yi2 = np.array([ 10, 11, 12,13,14,15,16,26, 28, 29,30,␣
 ↪29,28,26,16,15,14,13,12,11,10], dtype='double');
```

```python
x2 = np.linspace(-15, 15, num=1001, endpoint=True)

#Chamada de função

y_la2 = lagrange_interp(xi2,yi2,x2)

#Plot da solução

plt.figure(figsize=(10,6),facecolor='white')
plt.grid(color='k', linestyle='-', linewidth=1)
plt.plot(x2,y_la2,label = 'Interpolação de Lagrange',linewidth = 2)
plt.scatter(xi2,yi2,label = 'Pontos',linewidth = 2, facecolor='red')
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('Interpolação de Newton')
plt.legend()
plt.show()
```

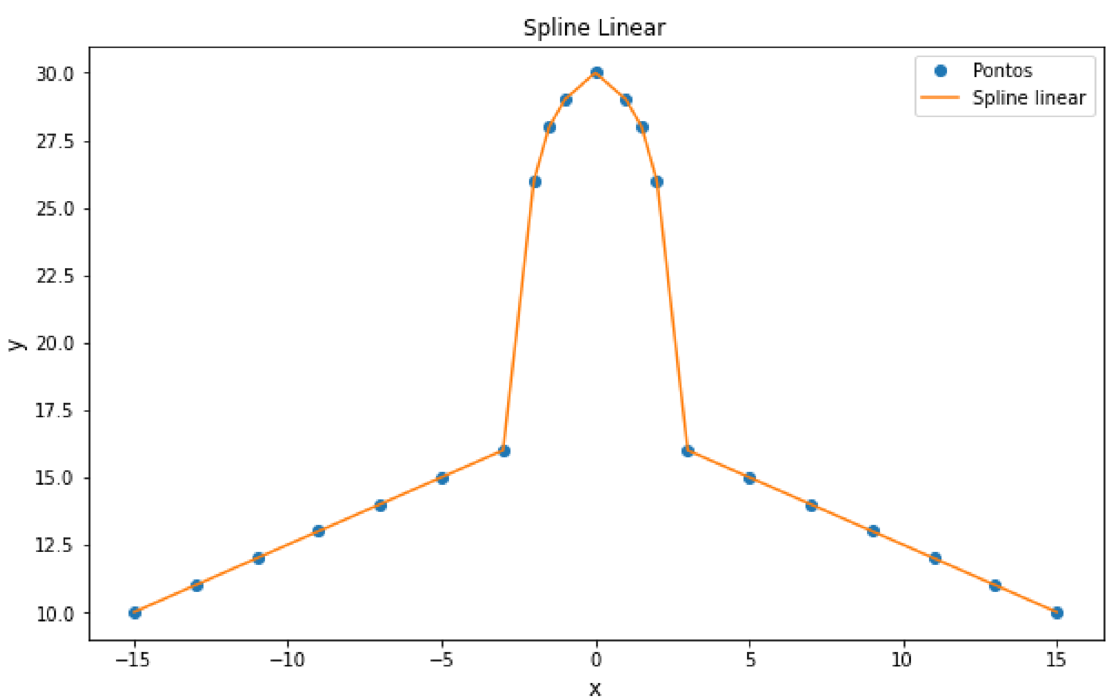


Observa-se que a figura obtida não se parece em nada com vista superior da aeronave, devido principalmente ao erro associado aos intervalos $-15 < x < -10$ e $10 < x < 15$, um ótimo exemplo do fenômeno de Runge.

Como 21 dados são fornecidos para a interpolação, é esperado que um grande erro seja verificado, visto a instabilidade que polinômios de graus maiores que 3 adquirem gradualmente, tendo como consequência, aumentando do erro da interpolação.

Para a spline linear, temos então:

[33]:
```
import numpy as np                #Import de bibliotecas
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d  #Import da função interpoladora

#Dados

xi2 = np.array([-15,-13,-11,-9,-7,-5,-3,-2,-1.5,-1, 0,1,1.5, 2, 3, 5, 7,␣
↪9,11,13,15], dtype='double');
yi2 = np.array([ 10, 11, 12,13,14,15,16,26, 28, 29,30,␣
↪29,28,26,16,15,14,13,12,11,10], dtype='double');

#Chamada da função de interpolação linear

f = interp1d(xi2, yi2)

xnew = np.linspace(-15, 15, num=1001, endpoint=True)

#plot

plt.figure(figsize=(10,6),facecolor='white')
plt.plot(xi2, yi2, 'o', xnew, f(xnew), '-')
plt.legend(['Pontos', 'Spline linear'], loc='best')
plt.xlabel('x',fontsize='large')
plt.ylabel('y',fontsize='large')
plt.title('Spline Linear')
plt.show()
```



Questão 4 (a)

O método dos gradientes consiste em resolver um sistema linear usando-se de um problema de minimização de uma função quadrática $F(\mathbf{x}) : R^n \to R$, sendo F expressa abaixo:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - b^T\mathbf{x} + c$$

Onde A é, necessariamente, uma matriz simétrica positiva, ou seja: $\mathbf{x}^T A\mathbf{x} > 0$.

Para a função acima, tem-se que seu ponto de mínimo $\mathbf{x}$ coincide com a solução do sistema linear:

$$A\mathbf{x} = b$$

A verificação desse fato pode ser feita através de ferramentas do cálculo, de modo que o gradiente da função $F(\mathbf{x})$ deve ser um vetor nulo e o determinante da matriz Hessiana deve ser positivo.

Antes de se aplicar o gradiente sobre a função, observa-se que, devido as operações matriciais, os elementos da imagem de $F(\mathbf{x})$ (escalares) são dados por:

$$f(\mathbf{x}) = \frac{1}{2}\sum_{i,j=1}^{n} a_{ij}x_i x_j - \sum_{i=1}^{n} b_i x_i + c$$

E, portanto, tem-se como gradiente:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n - b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n - b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n - b_n \end{bmatrix} = A\mathbf{x} - b = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Ou seja, $\mathbf{x}$ é ponto crítico. Porém ainda é necessário verificar se é ponto mínimo. Dessa forma, obtém-se a matriz Hessiana:

$$|\mathbf{H}| = \begin{vmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 x_2} & \dots & \frac{\partial^2 f(x)}{\partial x_1 x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \dots & \frac{\partial^2 f(x)}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n x_1} & \frac{\partial^2 f(x)}{\partial x_n x_2} & \dots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = |A|$$

Acima, verifica-se porque A deve ser simétrica positiva definida, onde sua simetria é necessaria pois $\frac{\partial^2 f(x)}{\partial x_i x_j} = \frac{\partial^2 f(x)}{\partial x_j x_i}$, além disso, por hipótese x é ponto de mínimo, o que é verificado quando A é positiva de finida, ou seja, temos que $|A| > 0$.

O processo iterativo tem como objetivo obter $\nabla F(\mathbf{x}) = 0$. Como se quer atingir o mínimo da função, segue-se o sentido de maior decrescimento da função, de forma que define-se o resíduo do processo:

$$-\nabla F(\mathbf{x}) = b - A\mathbf{x}^k = r^k$$

De modo que $\mathbf{x}^{k+1}$ é obtido como:

$$\mathbf{x}^{k+1} = x_k - \alpha \nabla F(\mathbf{x}^k) = \mathbf{x}^k + \alpha r^k$$

E, por sua vez, o parâmetro α é dado por:

$$\alpha = \frac{r^k . r^k}{r^k . A r^k}$$

Da própria expressão anterior, é possível concluir que resíduos resultantes de iterações consecultivas são perpendiculares, ou seja $r^{k+1} . r^k = 0$.

Questão 4 (b)

Escolhendo-se uma matriz de ordem três por três simétrica positiva definida, tem-se:

$$A = \begin{pmatrix} 10 & 1 & 0 \\ 1 & 10 & 1 \\ 0 & 1 & 10 \end{pmatrix}$$

A fim de verificar se essa matriz é SPD, aplica-se a verificação por determinantes principais, onde (por álgebra linear) é análogo afirmar que A é definida positiva se apresentar todos autovalores maiores que zero e determinantes principais maiores que zero. Nota-se fácilmente que é simétrica e, portanto, verifica-se se os determinantes principais da matriz são positivos:

$$|\,10\,| > 0$$

$$\begin{vmatrix} 10 & 1 \\ 1 & 10 \end{vmatrix} = 99 > 0$$

e, finalmente:

$$\begin{vmatrix} 10 & 1 & 0 \\ 1 & 10 & 1 \\ 0 & 1 & 10 \end{vmatrix} = 80 > 0$$

A matriz é SPD. Em seguida, define-se b como:

$$\mathbf{b} = \begin{pmatrix} 11 \\ 11 \\ 1 \end{pmatrix}$$

```
import numpy as np              #import de bibliotecas
from numpy import linalg as LA

def is_pos_def(x):          #verificação se A é SPD
    """verifique se uma matriz é simétrica positiva definida"""
    return np.all(np.linalg.eigvals(x) > 0) #verificação através de autovalores

def metodos_dos_gradientes(A, b, x, kmax):
    """
    Resolve Ax = b
    Parametro x: Valores inciais
    """
    if (is_pos_def(A) == False) | (A != A.T).any(): #Caso não seja SPD
        raise ValueError('A matriz A precisa ser simétrica positiva definida(SPD)')
    r = b - A @ x
    k = 0; erro = np.inf;
    while (erro > 1e-3 and k < kmax):  #processo iterativo e critério de parada
        p = r
        q = A @ p
        alpha = (p @ r) / (p @ q)
        x = x + alpha * p
        r = r - alpha * q
        k = k + 1
        erro = LA.norm(r)
        print('Iteração: %.d | Norma do erro: %.4f | x: ' %(k,erro)) #print a cada iteração
        print(x); print('\n')
    return x


#Dados do sistema
```

```
A = np.array([[10,1,0], [1,10,1],[0,1,10]])
b = np.array([11,11,1])
x0 = np.array([0,0,0])
kmax = 1000       # Critério de parada

#Chamada da função

metodos_dos_gradientes(A, b, x0, kmax)
```

```
Iteração: 1 | Norma do erro: 0.8983 | x:
[0.9922049  0.9922049  0.09020045]


Iteração: 2 | Norma do erro: 0.0808 | x:
[1.00077186e+00 9.91759863e-01 8.59247324e-04]


Iteração: 3 | Norma do erro: 0.0114 | x:
[1.00082399e+00 9.99833507e-01 8.24028484e-04]


Iteração: 4 | Norma do erro: 0.0016 | x:
[1.00001631e+00 9.99835199e-01 1.63109850e-05]


Iteração: 5 | Norma do erro: 0.0002 | x:
[1.00001648e+00 9.99996670e-01 1.64800622e-05]


array([1.00001648e+00, 9.99996670e-01, 1.64800622e-05])
```

```
from google.colab import drive
drive.mount("/content/gdrive", force_remount=True)
```

```
Mounted at /content/gdrive
```

```
%%capture
!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('WESLEYGONÇALVESDASILVA_11233140_AERONÁUTICA_2.ipynb')
```