Universidad Central de Venezuela

Facultad de Ingeniería

Escuela de Ingeniería Eléctrica

Laboratorio N°2

Periodo 2023-3

06/12/2023

CI 29571461 Ricardo Santana

1. Introducción

A través de la interpolación, es posible obtener una función que se adapte, con cierto margen de error, a un conjunto de datos. Esto nos permite modelar el comportamiento de un fenómeno o sistema que deseamos controlar o del cual queremos llevar un registro.

Es de gran utilidad conocer diversos métodos numéricos que permitan realizar la interpolación de manera más efectiva, según su aplicación. Para esto, se utilizan polinomios interpolantes. En el siguiente informe se explican los métodos de interpolación a través del polinomio de Lagrange, el polinomio de Newton, los polinomios trigonométricos, así como el ajuste por mínimos cuadrados.

2. Marco teórico

El problema de determinar un polinomio de grado uno que pasa por diferentes puntos (x_0, y_0) y (x_1, y_1) es igual al de aproximar una función f para la que $f(x_0) = y_0$ y $f(x_1) = y_1$ por medio de un polinomio de primer grado que se interpola, o que coincida con los valores de f en los puntos determinados. El uso de estos polinomios para aproximación dentro del intervalo determinado mediante puntos finales recibe el nombre de interpolación. [1]

Según el libro de texto [2] se pueden ajustar funciones de interpolación a una lista de datos siguiendo dos criterios: ajuste exacto y ajuste por mínimos cuadrados. Decidir el tipo de ajuste a utilizar, depende de las necesidades de la investigación u origen de los datos.

2.1 Ajuste Exacto [2]

Una de las técnicas de interpolación (exacta) para un conjunto de datos $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ es la interpolación polinomial, donde se busca un polinomio $p(x) = a_0 + a_1 x + a_2 x_2 + \cdots + a_n x_n$ de menor grado posible que contenga a todos los datos conocidos, es decir, $p(x_i) = y_i$ para todo $i = 0, 1, \ldots, n$. Por ejemplo, para los datos (1,−1), (2,−4) y (3,−9), el interpolador polinomial es $p(x) = -x^2$ dado que no existe un polinomio de menor grado que contenga a estos puntos. A continuación se describiran tres metodos distintos para construir un polinomio interpolante.

2.1.1 Polinomio de interpolación de Lagrange [1]

Si $x_0, x_1, \ldots, x_n$ son n + 1 números distintos y f es una función cuyos valores están determinados en estos números, entonces existe un único polinomio $P(x)$ de grado a lo sumo n con

$$f(x_k) = P(x_k), k = 0, 1, \ldots, n.$$

Este polinomio está determinado por

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x),$$

donde, para cada k = 0, 1, . . . , n,

$$L_{n,k}(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)} = \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{(x - x_i)}{(x_k - x_i)}$$

2.1.2 Polinomio de interpolación de Newton [2]

En la propuesta de Newton, primero se hace pasar un polinomio de grado cero por uno de los puntos, y desde el anterior se construye un polinomio de grado uno que pasa por otro punto de la lista. Desde los dos últimos puntos, se construye un polinomio de grado dos que pasa por un tercer punto de la lista y así sucesivamente. De esta manera, se respeta el trabajo anterior, y no se tienen dificultades para agregar puntos a la lista.

El polinomio $P_0(x)$ se construye de manera que pase exactamente por el punto (x_0, y_0), de donde la propuesta más simple es $P(x) = P_0(x) = y_0$

Ahora se construye un nuevo polinomio que pase por (x_1, y_1), pero que parta de $P_0(x)$. Este nuevo polinomio de grado uno tiene la forma $P_1(x) = P_0(x) + c_1(x - x_0)$. Como $P_1(x)$ debe interpolar a (x_1, y_1), al evaluar en x_1 debe dar como resultado y_1 y por tanto: $P_1(x_1) = y_0 + c_1(x_1 - x_0) = y_1$. Al despejar c_1 se obtiene:

$$c_1 = \frac{y_1 - y_0}{x_1 - x_0},$$

expresión conocida como diferencia dividida de orden uno y se denota como:

$$f[x0, x1] = \frac{y_1 - y_0}{x_1 - x_0}.$$

Para unificar la notación, decimos que $y_0 = f[x_0]$ es una diferencia dividida de orden cero, y el polinomio de interpolación con dicha notación sería:

$$P_1(x) = f[x_0] + f[x_0, x_1](x - x_0).$$

De igual forma, se construye un polinomio que capture otro punto más, es decir, que pase por (x_0, y_0), por (x_1, y_1) y por (x_2, y_2) sin perder el trabajo que se ha hecho hasta el momento. Dado $P_2(x) = f[x_0] + f[x_0, x_1](x - x_0) + c_2(x - x_0)(x - x_1)$, notar que al evaluar en x0, los dos últimos términos se anulan, quedando solamente $P_0(x)$, cuyo valor es y_0. Igualmente, al evaluar en x_1, el último término se anula, quedando el polinomio $P_1(x)$ ya calculado, cuyo valor es y_1 cuando se evalúa en x_1. Para encontrar el coeficiente c_2, se debe tener en cuenta que, evaluar $P_2(x)$ en x_2 resulte en y_2. Al hacer lo anterior, se obtiene:
$P_2(x_2) = f[x_0] + f[x_0, x_1](x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) = y_2$. Al despejar y ordenar adecuadamente se obtiene

$$c_2 = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}$$

Notar que en el numerador hay diferencias divididas de orden uno y escribiendo en notación de diferencias, se tiene

$$c2 = f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0},$$

expresión llamada diferencia dividida de orden dos. El polinomio de interpolación en notación de diferencias divididas quedaría

$$P_2(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1).$$

Ahora, de manera general, cuando se tienen n + 1 puntos, el polinomio se puede escribir como

$$\begin{aligned} P(x) = f[x_0] + f[x_0, x_1](x - x_0) &+ f[x_0, x_1, x_2](x - x_0)(x - x_1) \\ &+ f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2) \\ &\vdots \\ &+ f[x_0, \ldots, x_n](x - x_0) \cdots (x - x_{n-1}), \end{aligned}$$

donde, en general, se tiene la siguiente expresión para una diferencia dividida de orden k:

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

2.2 Ajuste por Mínimos Cuadrados [2]

Si lo que se desea es marcar una tendencia, los polinomios de ajuste exacto no son los más adecuados. Es mejor buscar una curva más simple, que tal vez no contiene la totalidad de los puntos, pero que pasa cerca de cada uno de ellos.

2.2.1 Errores

- **Error relativo**: suponer que se tienen n+1 puntos puntos $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ y que estos se ajustan con la función $\overline{f}(x)$. El error relativo que se comete con la funciónn de ajuste es $E_r = \sum_{i=0}^{n}(y_i - \overline{f}(x_i))$. Un problema en esta situación es de compensación, pues pueden existir errores grandes que al sumarlos con otros de igual magnitud pero de signo contrario, se cancelen y pareciera que se tiene un pequeño o inclusive nulo error.
- **Error absoluto**: para evitar que el error se compense, se toma el valor absoluto y elerror se puede calcular como $E_a = \sum_{i=0}^{n} |y_i - \overline{f}(x_i)|$. Este error presenta dos problemas. Primero, puede ser que una curva ajuste bien una lista de puntos, pero la suma de errores pequeños en cada punto puede finalmente arrojar un error grande, y segundo, el hecho que la función valor absoluto presente problemas de diferenciabilidad.
- **Error cuadrático**: esta manera de medir el error tiene la virtud de no presentar problemas de diferenciabilidad y se define como $E_a = \sum_{i=0}^{n}(y_i - \overline{f}(x_i))^2$. Además, para valores $|yi - \overline{f}(x_i)| < 1$ se tiene $(y_i - \overline{f}(x_i))^2 < |yi - \overline{f}(x_i)|$, y por tanto errores “pequeños” se transforman en valores inclusive menores.

2.2.2 Polinomios de mínimos cuadrados

Cuando se usa la familia de los monomios $\{1, x, x^2, \ldots\}$, combinaciones lineales de elementos de esta base producen un polinomio, y el caso más simple es cuando se usan los primeros dos elementos de la base, obteniéndose polinomios de grado uno. Ahora, la idea entonces es encontrar una recta $a_0 + a_1 x$ que pase cerca de los puntos $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. Si se observa el error cuadrático E entre la curva $\overline{f}(x) = a_0 + a_1 x$ y la lista de puntos, dicha curva presenta un error cuadrático dado por la función de dos variables

$$E(a_0, a_1) = \sum_{i=0}^{n} (y_i - a_0 - a_1 x_i)^2.$$

Ahora la intención es hacer mínimo dicho error, así que hay que buscar dicho valor usando las técnicas usuales del cálculo vectorial. Para tal efecto, notar que $E(a_0, a_1)$ es una función cuadrática y su gráfica corresponde a un paraboloide que se abre hacia arriba. Luego, tiene un único punto crítico y corresponde a su mínimo absoluto. Para calcular el punto crítico, se debe determinar cuando el gradiente

$$\bigtriangledown E = \frac{\partial E}{\partial a_0} i + \frac{\partial E}{\partial a_1} j$$

de la función E es el vector nulo. En tal caso se obtienen las dos ecuaciones normales

$$\frac{\partial E}{\partial a_0} = 0, \frac{\partial E}{\partial a_1} = 0$$

A continuación se presenta el cálculo detallado de la primera ecuación.

$$\frac{\partial E}{\partial a_0} = \frac{\partial}{\partial a_0}(\sum_{i=0}^{n} (y_i - a_0 - a_1 x_i)^2) = 0$$

$$\sum_{i=0}^{n} \frac{\partial}{\partial a_0} (y_i - a_0 - a_1 x_i)^2 = 0$$

$$\sum_{i=0}^{n} 2(y_i - a_0 - a_1 x_i)(-1) = 0$$

$$(-2) \left\{ \sum_{i=0}^{n} (y_i - a_0 - a_1 x_i) \right\} = 0$$

$$\sum_{i=0}^{n} (y_i - a_0 - a_1 x_i) = 0$$

$$\sum_{i=0}^{n} y_i - \sum_{i=0}^{n} a_0 - \sum_{i=0}^{n} a_1 x_i = 0$$

$$a_1 \sum_{i=0}^{n} x_i + a_0 (n+1) = \sum_{i=0}^{n} y_i,$$

la cual corresponde a la primera ecuación normal. Razonando de la misma manera, pero derivando E respecto a a_1, se llega a

$$a_1 \sum_{i=0}^{n} x_i^2 + a_0 \sum_{i=0}^{n} x_i = \sum_{i=0}^{n} y_i x_i$$

Al tener las dos ecuaciones normales, se llega a un sistema de ecuaciones cuya matriz aumentada es

$$\begin{pmatrix} \sum_{i=0}^{n} x_i^2 & \sum_{i=0}^{n} x_i & | & \sum_{i=0}^{n} y_i x_i \\ \sum_{i=0}^{n} x_i & n+1 & | & \sum_{i=0}^{n} y_i \end{pmatrix}$$

El sistema tiene solución única. Al usar la regla de Cramer se desprende que

$$a_1 = \frac{\begin{vmatrix} \sum_{i=0}^{n} y_i x_i & \sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i & n+1 \end{vmatrix}}{\begin{vmatrix} \sum_{i=0}^{n} x_i^2 & \sum_{i=0}^{n} x_i \sum_{i=0}^{n} x_i & n+1 \end{vmatrix}} = \frac{(n+1) \sum_{i=0}^{n} y_i x_i - \sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i}{(n+1) \sum_{i=0}^{n} x_i^2 - (\sum_{i=0}^{n} x_i)^2}$$

$$a_0 = \frac{\begin{vmatrix} \sum_{i=0}^{n} x_i^2 & \sum_{i=0}^{n} y_i x_i \sum_{i=0}^{n} x_i & \sum_{i=0}^{n} y_i \end{vmatrix}}{\begin{vmatrix} \sum_{i=0}^{n} x_i^2 & \sum_{i=0}^{n} x_i \sum_{i=0}^{n} x_i & n+1 \end{vmatrix}} = \frac{\sum_{i=0}^{n} x_i^2 \sum_{i=0}^{n} y_i - \sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i x_i}{(n+1) \sum_{i=0}^{n} x_i^2 - (\sum_{i=0}^{n} x_i)^2}$$

Finalmente, estos son los valores de a_0 y a_1 que hacen mínimo el error cuadrático.

2.3 Aproximación polinomial trigonométrica

2.3.1 Transformada de Fourier

Según la definifión 1 en [3] La transformada de Fourier (TF) de una función $f : \mathbb{R} \to \mathbb{C}$ es una funcion definida por

$$\mathbb{F}[f(t)] = F(j\mathrm{w}) = \int_{-\infty}^{\infty} f(t) e^{-j\mathrm{w}t} dt$$

2.3.2 Polinomios trigonométricos ortogonales [1]

Sea τ_n el conjunto de todas las combinaciones lineales de las funciones $\phi_0, \phi_1, \ldots, \phi_{2n-1}$. Este conjunto recibe el nombre de conjunto de polinomios trigonométricos de grado menor o igual a n. (Algunas fuentes también incluyen una función adicional en el conjunto $\phi_{2n}(x) = sen(nx)$.)

Para una función $f \in C[-\pi, \pi]$, queremos encontrar la aproximación de mínimos cuadrados continuos mediante las funciones en τ_n de la forma

$$S_n(x) = \frac{a_0}{2} + a_n cos(nx) + \sum_{k=1}^{n-1} (a_k cos(kx) + b_k sen(kx))$$

La selección adecuada de coeficientes es

$$a_k = \frac{\int_{-\pi}^{\pi} f(x)cos(kx)dx}{\int_{-\pi}^{\pi} cos(kx)^2 dx} = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)cos(kx)dx, k = 0, 1, 2, \ldots, n,$$

y

$$b_k = \frac{\int_{-\pi}^{\pi} f(x)sen(kx)dx}{\int_{-\pi}^{\pi} sen(kx)^2 dx} = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)sen(kx)dx, k = 0, 1, 2, \ldots, n.$$

El límite de $S_n(x)$ cuando $n \to \infty$ recibe el nombre de serie de Fourier de f. Las series de Fourier se usan para describir la solución de las diferentes ecuaciones ordinarias y diferenciales parciales que se presentan en situaciones físicas.

3. Práctica

Clase *data* que incluye metodos para obterner polinomios interpolantes de un conjunto de datos:

```
#importación de librerias a utilizar
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
import pandas as pd
```

```
class data():
    def __init__(self, abscisas, ordenadas):
        self.x = np.array(abscisas)
        self.y = np.array(ordenadas)
    #herramientas
    def error(self, y, fx):
        error = 0
        for i, j in zip(y, fx):
            error += (i-j)**2
        return error

    def graficar_pol_sp(self, polinomio, titulo):
        plt.grid()
        plt.plot(self.x,self.y,'bo')

        d = np.linspace(min(self.x),max(self.x))
        f = [polinomio.subs(x,i) for i in d]
        plt.plot(d,f,'r-')

        plt.title(titulo)
        plt.ylabel('Y')
        plt.xlabel('X')
        plt.show()

    def graficar_pol_np(self, polinomio, titulo):
        plt.grid()
        plt.plot(self.x,self.y,'bo')

        d = np.linspace(min(self.x),max(self.x),100)
        plt.plot(d,polinomio(d),'r-')

        plt.title(titulo)
        plt.ylabel('Y')
        plt.xlabel('X')
        plt.show()
    #metodos
    def polinomio_coef_ind(self):
        m1 = len(self.x) #, n1
        m2 = len(self.y) #, n2
        if (m1 != m2): # or (n1 != n2):
            raise ValueError('x e y no tienen la misma dimensión')

        m = m1
        n = m - 1
        B = np.ones((m, m)) # inicializar una matriz con unos

        # completar las m-1 filas de B, con las potencias apropiadas de los valores
        # que corresponden a la variable independiente.
        for i in range(m):
            for j in range(1, m):
                B[i, j] = self.x[i]**(j)

        c = np.linalg.solve(B, self.y.reshape((m2,1))) # resuelve el sistema de ecuaciones, dando como resultado los coefi

        c = np.flip(c).T  #invirtiendo coeficientes (agregado)
```

```
    p = np.poly1d(c[0]) # librería para trabajo simbólico de polinomios

    return c, p

def polinomio_lagrange(self):
    x = sp.symbols('x')

    n = len(self.x)
    p = 0
    for k in range(n):
        d = 1
        for i in range(n):
            if i != k:
                d = d * (x - self.x[i]) / (self.x[k] - self.x[i])
        p += self.y[k] * d

        p = sp.simplify(p) #simplificando a su minima expresion el polinomio de lagrage
    return p

def polinomio_newton(self):
    if len(self.x) != len(self.y):
        raise ValueError('Error en el tamaño de la data')

    n = len(self.x)
    F = [[0] * n for _ in range(n)]
    for i in range(n):
        F[i][0] = self.y[i]

    for i in range(1, n):
        for j in range(1, i + 1):
            F[i][j] = (F[i][j - 1] - F[i - 1][j - 1]) / (self.x[i] - self.x[i - j])

    coeficientes = [F[i][i] for i in range(n)]

    #creando polinomio
    ecuacion = 0
    termino = 1
    var = sp.symbols('x')
    for k in range(n):
        ecuacion += coeficientes[k]*termino
        termino *= (var - self.x[k])

    p = sp.simplify(ecuacion)

    return F, p

def polinomio_min_cuad(self, G):
    n = len(self.x)
    B = np.zeros(G + 1)
    F = np.zeros((n, G + 1))

    for k in range(1, G + 2):
        F[:, k - 1] = self.x ** (k - 1)

    A = np.dot(F.T, F)
    B = np.dot(F.T, self.y)
    C = np.linalg.solve(A, B)

    c = np.flip(C).T  #invirtiendo coeficientes (agregado)
    #polinomio
    p = np.poly1d(c)

    return c, p

def __coeficientes_poli_trig(self, G):
    X = self.x #copias
    Y = self.y

    n = len(X) - 1
    maxg = int((n - 1) / 2)
    if G > maxg:
        G = maxg

    A = np.zeros(G + 1)
    B = np.zeros(G + 1)
    Yes = (Y[0] + Y[n]) / 2
    Y[0] = Yes
    Y[n] = Yes
    A[0] = np.sum(Y)
```

```
        for i in range(1, G + 1):
            A[i] = np.dot(np.cos(i * X), Y.T)
            B[i] = np.dot(np.sin(i * X), Y.T)

        A = 2 * A / n
        B = 2 * B / n
        A[0] = A[0] / 2

        return A, B

    def polinomio_trigonometrico(self, G):
        A, B = self.__coeficientes_poli_trig(G)
        x = sp.symbols('x')
        f = A[0]
        for k in range(1, G + 1):
            f += A[k] * sp.cos(k * x) + B[k] * sp.sin(k * x)
```

3.1 Problema test

Tabla 1. data problema test

X	Y
0.7	0.5428
1.0	0.7652
1.3	0.6893
1.6	0.1946
1.9	0.2818
2.2	0.1104

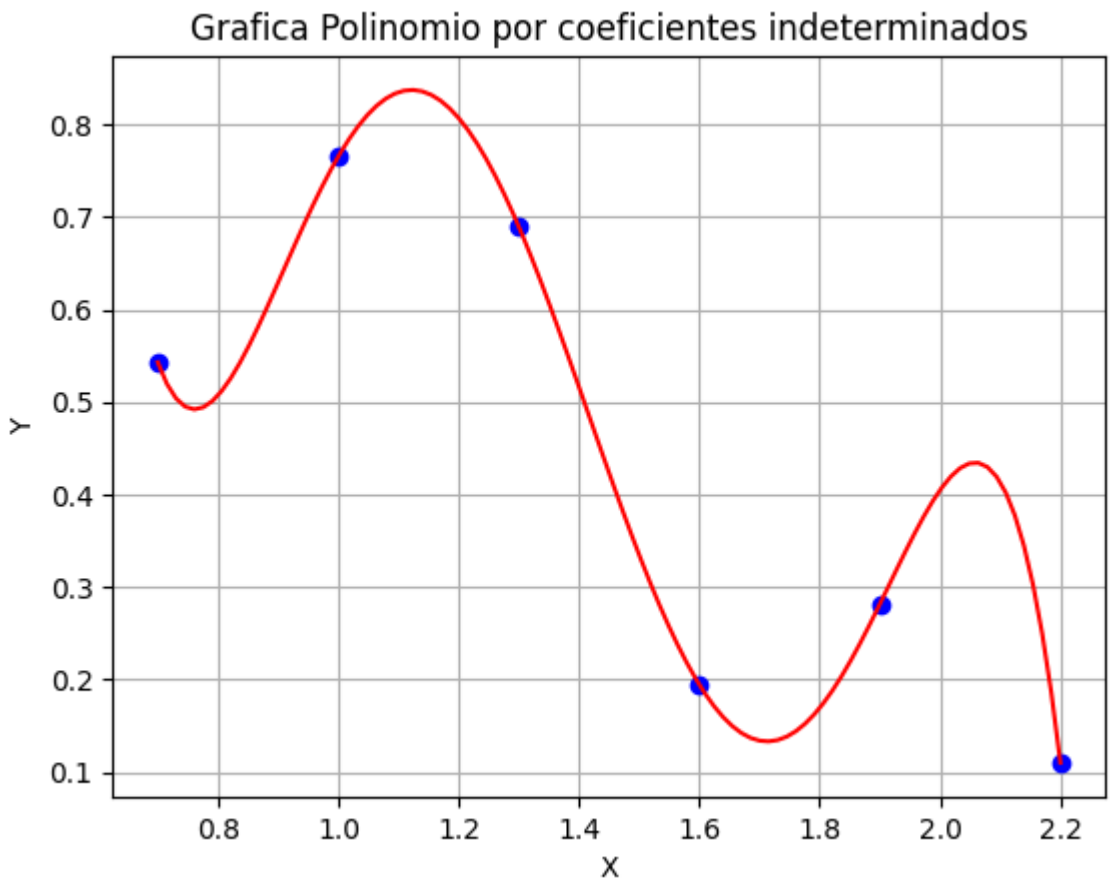
```
#data problema test
x1 = [0.7, 1.0, 1.3, 1.6, 1.9, 2.2]
y1 = [0.5428, 0.7652, 0.6893, 0.1946, 0.2818, 0.1104]
data1 = data(x1, y1)
x = sp.symbols('x')
```

3.1.1 Polinomio interpolante por coeficientes indeterminados

```
coe1, pol1 = data1.polinomio_coef_ind()
print(pol1)
```

```
             5        4         3         2
     -10.16 x + 71.8 x - 194.4 x + 250.4 x - 152.9 x + 36.04
```

```
#graficando
data1.graficar_pol_np(pol1, 'Grafica Polinomio por coeficientes indeterminados')
```



Observación: se estudirá en cada caso el error cuadrático de cada aproximación.

```
error1 = data1.error(y1,pol1(x1))
error1
```

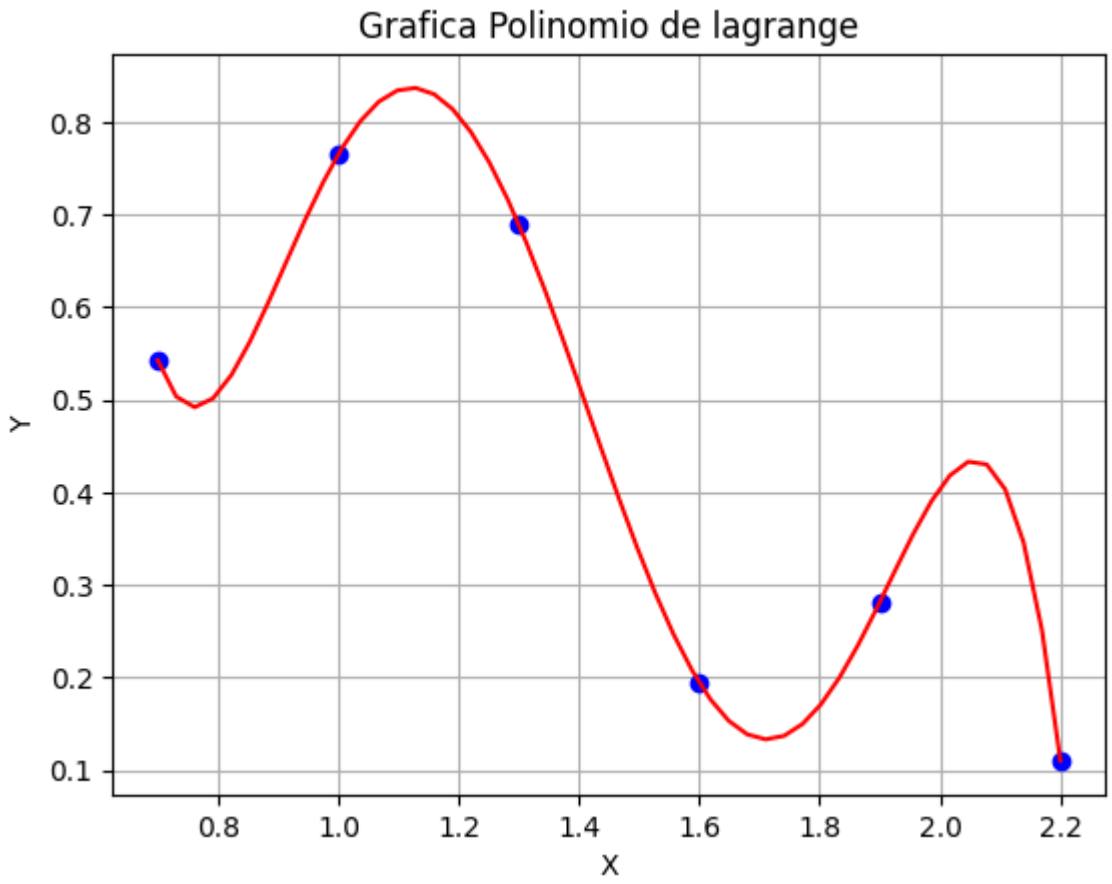
```
4.450423446420711e-25
```

3.1.2 Polinomio interpolante por Lagrange

```
pol2 = data1.polinomio_lagrange()
pol2
```

$$-10.159122085048x^5 + 71.8017832647463x^4 - 194.391838134431x^3 + 250.408257887517x^2 - 152.936235116598x + 36.0423541838135$$

```
#graficando
data1.graficar_pol_sp(pol2, 'Grafica Polinomio de lagrange')
```




```
error2 = data1.error(y1, (pol2.subs(x,i) for i in x1))
error2
```

$$1.07973616315574 \cdot 10^{-24}$$

3.1.3 Polinomio interpolante de Newton

```
difdiv3, pol3 = data1.polinomio_newton()
pol3
```

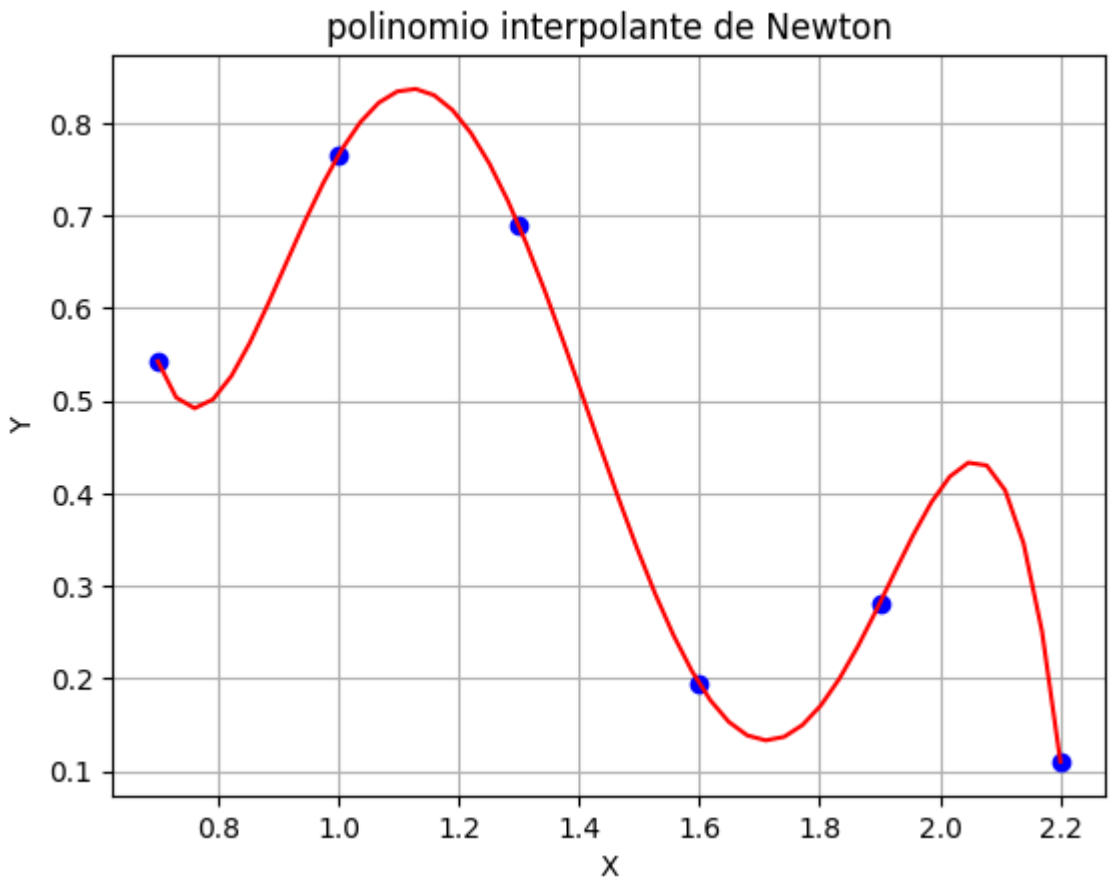
$$-10.159122085048x^5 + 71.8017832647462x^4 - 194.391838134431x^3 + 250.408257887517x^2 - 152.936235116598x + 36.0423541838134$$

```
df3 = pd.DataFrame(difdiv3)
print('tabla diferencias divididas')
df3
```

tabla diferencias divididas

	0	1	2	3	4	5
0	0.5428	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.7652	0.741333	0.000000	0.000000	0.000000	0.000000
2	0.6893	-0.253000	-1.657222	0.000000	0.000000	0.000000
3	0.1946	-1.649000	-2.326667	-0.743827	0.000000	0.000000
4	0.2818	0.290667	3.232778	6.177160	5.767490	0.000000
5	0.1104	-0.571333	-1.436667	-5.188272	-9.471193	-10.159122

```
#graficando
data1.graficar_pol_sp(pol3, 'polinomio interpolante de Newton')
```




```
error3 = data1.error(y1, (pol3.subs(x,i) for i in x1))
error3
```

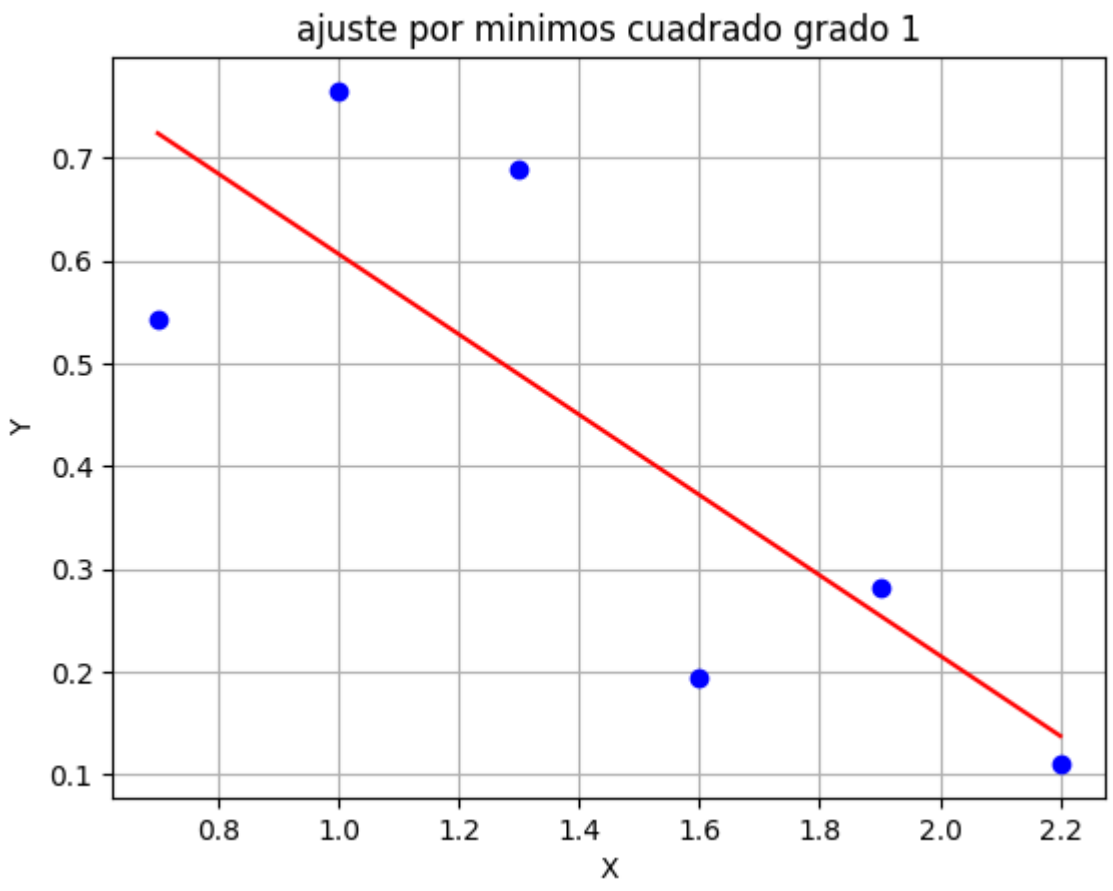
$$6.62573894700386 \cdot 10^{-25}$$

3.1.4 Ajuste por minimos cuadrados

```
coe4, pol4 = data1.polinomio_min_cuad(1)
print(pol4)
```

```
-0.3911 x + 0.9978
```

```
data1.graficar_pol_np(pol4, 'ajuste por minimos cuadrado grado 1')
```



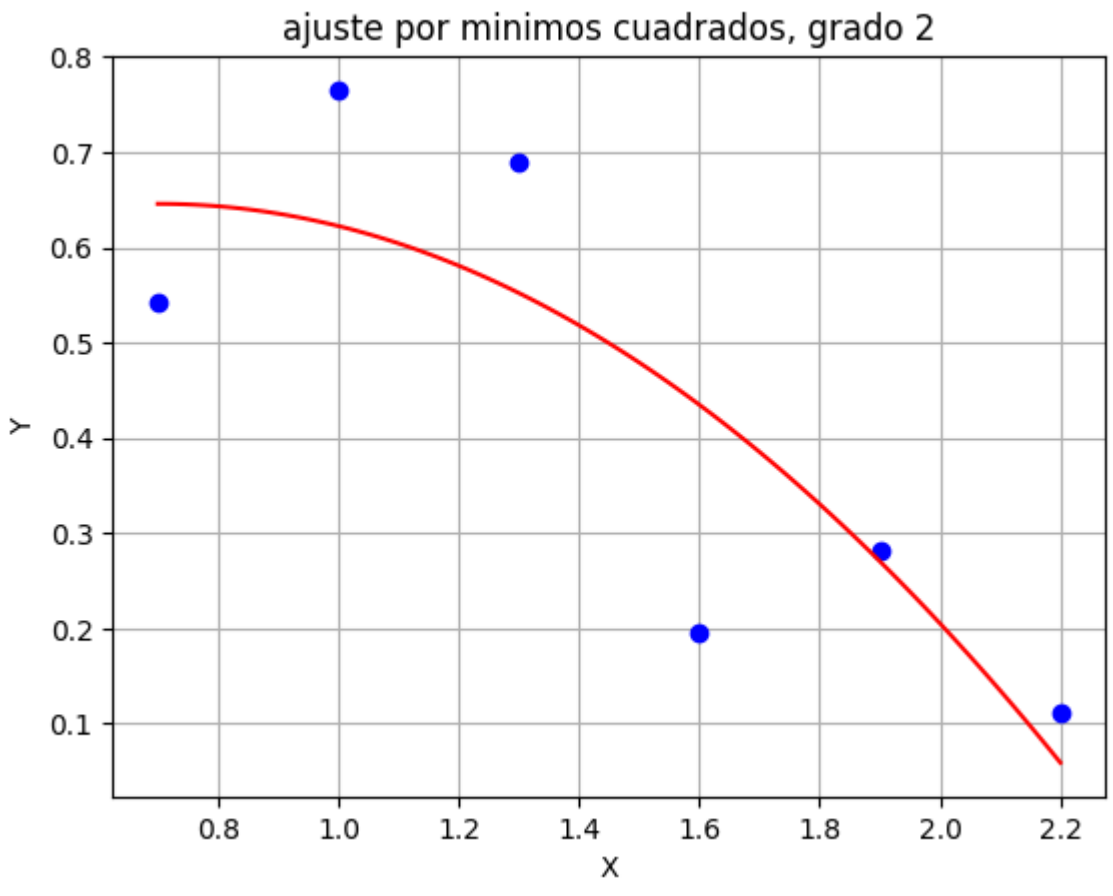

```
error4 = data1.error(y1, pol4(x1))
error4
```

```
0.1308853053333333
```

```
coe5, pol5 = data1.polinomio_min_cuad(2)
print(pol5)
```

```
             2
-0.2612 x + 0.3664 x + 0.5172
```

```
data1.graficar_pol_np(pol5, 'ajuste por minimos cuadrados, grado 2')
```




```
error5 = data1.error(y1, pol5(x1))
error5
```

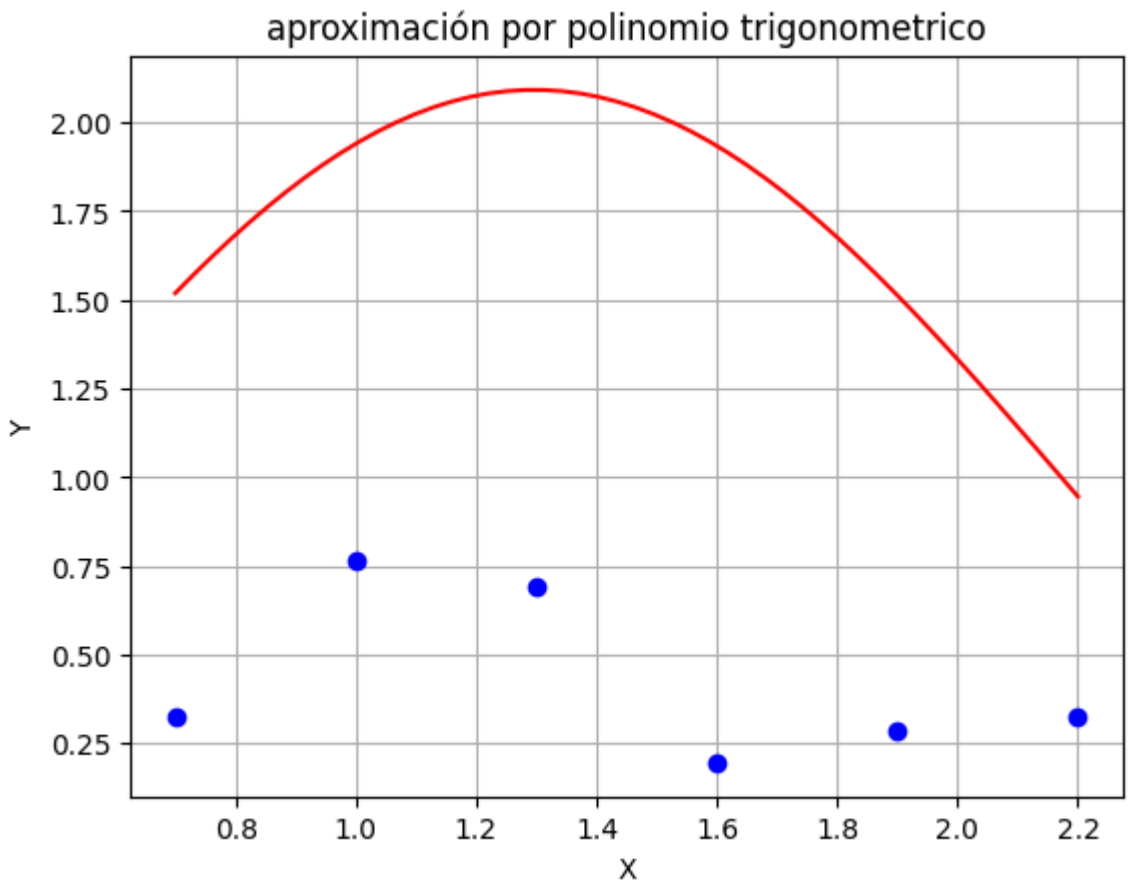
```
0.11024916771428571
```

3.1.5 Aproximación trigonométrica

```
pol6 = data1.polinomio_trigonometrico(2)
pol6
```

$$0.897486025913829 \sin(x) + 0.351361187642719 \sin(2x) + 0.223453613404312 \cos(x) - 0.548446195229585 \cos(2x) + 0.51682$$

```
data1.graficar_pol_sp(pol6, 'aproximación por polinomio trigonometrico')
```




```
error6 = data1.error(y1, (pol6.subs(x,i) for i in x1))
error6
```

$$9.54175953539958$$

El error establecido por el método trigonometrico es alto, por tanto no se tomará en cuenta para proximos estudios.

3.2 Ejercicio práctico 1

Tabla 2. data ejercicio práctico 1

X	Y
0.0	-1.000
1.5	-3.625
3.0	5.000
4.5	45.125
6.0	137.000
7.5	300.875
9.0	557.000
10.5	925.625
12.0	1427.000
13.5	2081.375
15.0	2909.000
16.5	3930.125
18.0	5165.000
19.5	6633.875

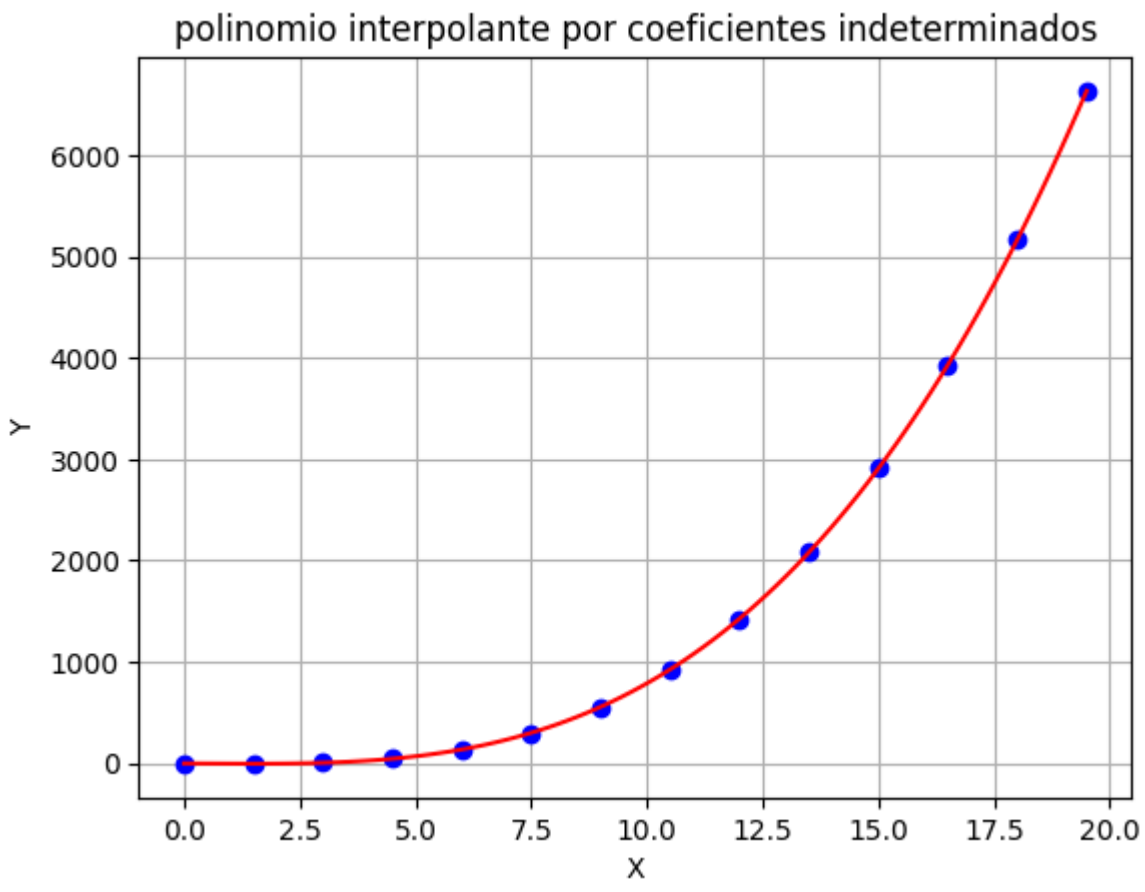
```
#data ejercicio practico
x2 = [0.0, 1.5, 3.0, 4.5, 6.0, 7.5, 9.0, 10.5, 12.0, 13.5, 15.0, 16.5, 18.0, 19.5]
y2 = [-1.000, -3.625, 5.000, 45.125, 137.000, 300.875, 557.000, 925.625, 1427.000, 2081.375, 2909.000, 3930.125, 5165.000,
data2 = data(x2, y2)
x = sp.symbols('x')
```

3.2.1 Polinomio interpolante por coeficientes indeterminados

```
coe7, pol7 = data2.polinomio_coef_ind()
print(pol7)
```

```
                13             12             11             10
     -2.994e-22 x  + 4.013e-20 x  - 2.396e-18 x  + 8.402e-17 x
                 9             8             7             6            5
      - 1.921e-15 x + 3.006e-14 x - 3.287e-13 x + 2.516e-12 x - 1.33e-11 x
                  4     3     2
      + 4.696e-11 x + 1 x - 2 x - 1 x - 1
```

```
data2.graficar_pol_np(pol7, 'polinomio interpolante por coeficientes indeterminados')
```




```
error1 = data2.error(y2, pol1(x2))
error1
```

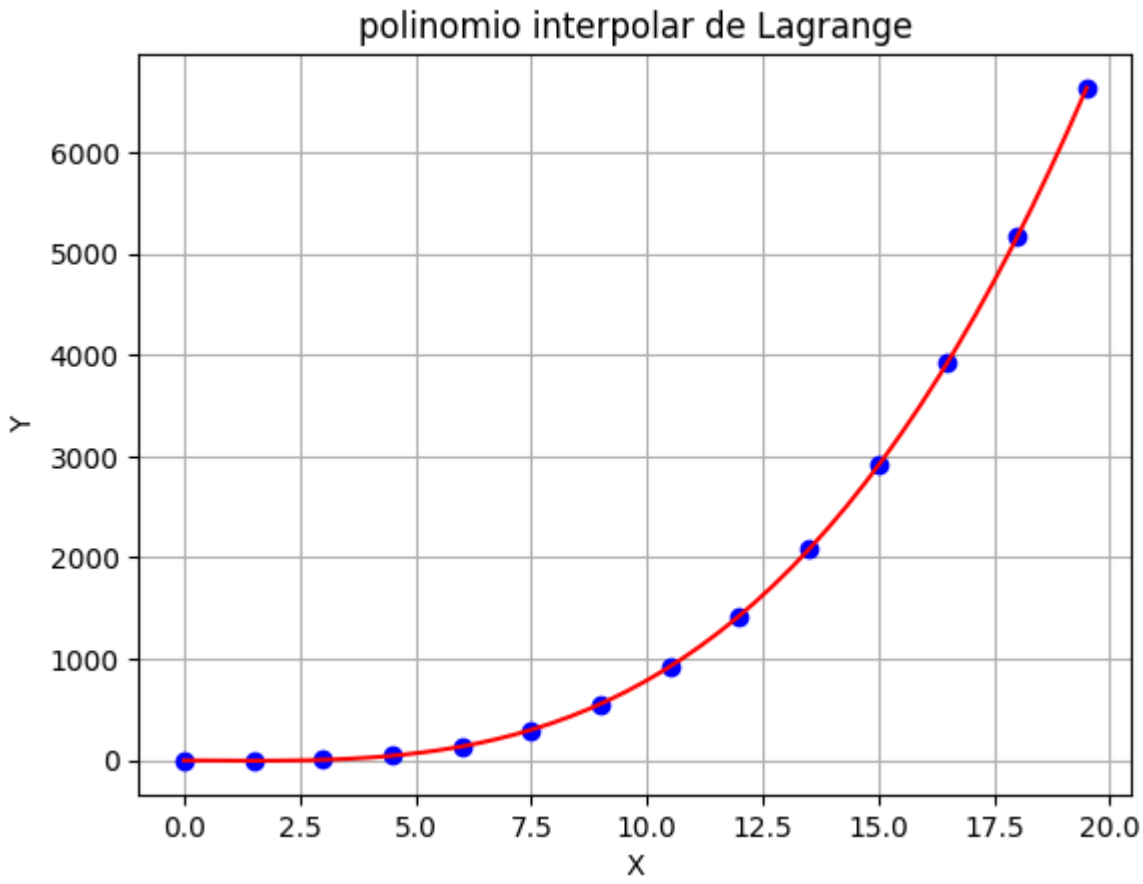
```
     640267040976973.0
```

3.2.2 Polinomio interpolante de Lagrange

```
pol8 = data2.polinomio_lagrange()
pol8
```

$$-4.11108764438855 \cdot 10^{-22}x^{13} + 5.6327690992411 \cdot 10^{-20}x^{12} - 2.49366499671666 \cdot 10^{-18}x^{11} + 6.91720986045752 \cdot 10^{-17}x^{10} - 1.79370407416002 \cdot 10^{-15}x^{9} + 3.56381590904675 \cdot 10^{-14}x^{8} - 2.33146835171283 \cdot 10^{-13}x^{7} + 2.81374923361 \cdot 10^{-12}x^{6} - 1.86162196769146 \cdot 10^{-12}x^{5} + 1.88151716429275 \cdot 10^{-11}x^{4} + 0.999999999993634x^{3} - 1.9999999999967x^{2} - 1.00000000003968x - 1.0$$

```
data2.graficar_pol_sp(pol8, 'polinomio interpolar de Lagrange')
```




```
error2 = data2.error(y2, (pol8.subs(x,i) for i in x2))
error2
```

$$7.55026582653273 \cdot 10^{-8}$$

3.2.3 Polinomio interpolante de Newton

```
difdiv9, pol9 = data2.polinomio_newton()
pol9
```

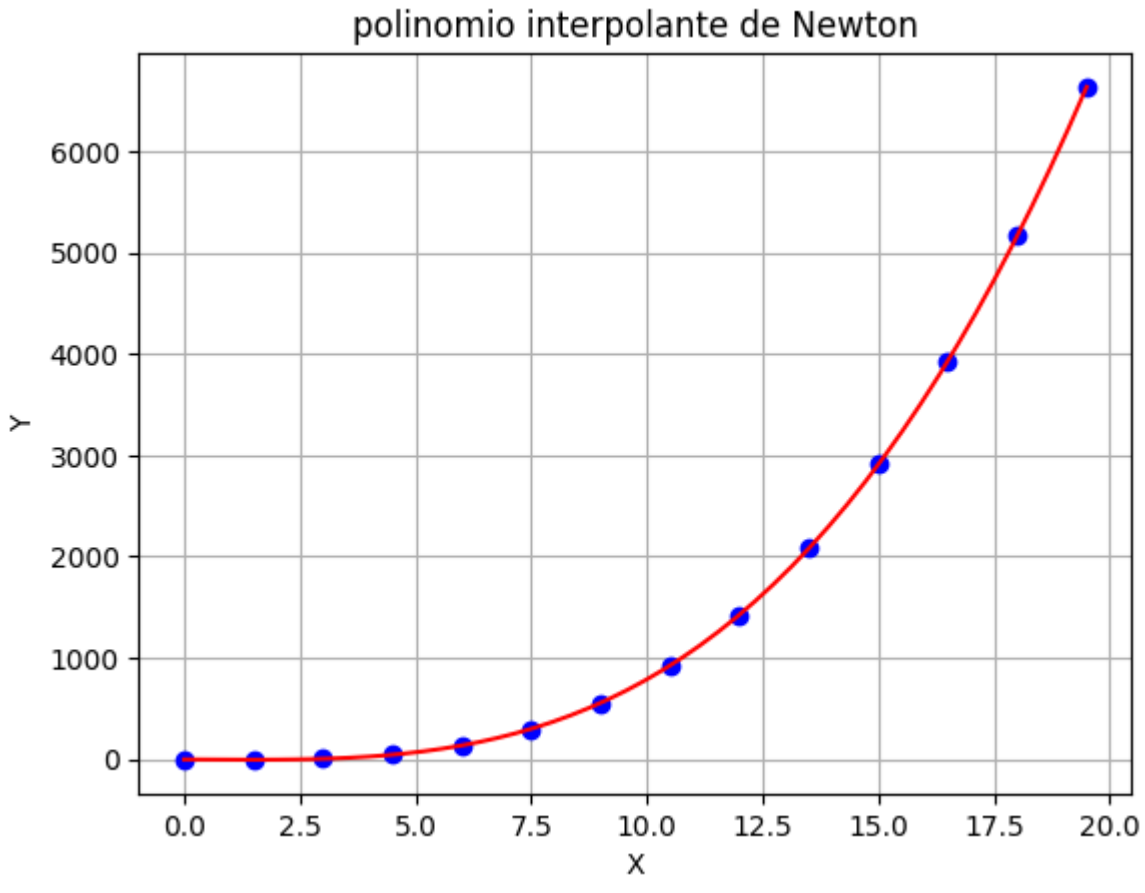
$$1.0x^{3} - 2.0x^{2} - 1.0x - 1.0$$

```
df9 = pd.DataFrame(difdiv9)
print('diferencias divididas')
df9
```

diferencias divididas

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	-1.000	0.00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	-3.625	-1.75	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	5.000	5.75	2.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	45.125	26.75	7.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	137.000	61.25	11.5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	300.875	109.25	16.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	557.000	170.75	20.5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	925.625	245.75	25.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	1427.000	334.25	29.5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	2081.375	436.25	34.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	2909.000	551.75	38.5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
11	3930.125	680.75	43.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	5165.000	823.25	47.5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13	6633.875	979.25	52.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
data2.graficar_pol_sp(pol9, 'polinomio interpolante de Newton')
```




```
error9 = data2.error(y2, (pol9.subs(x,i) for i in x2))
error9
```

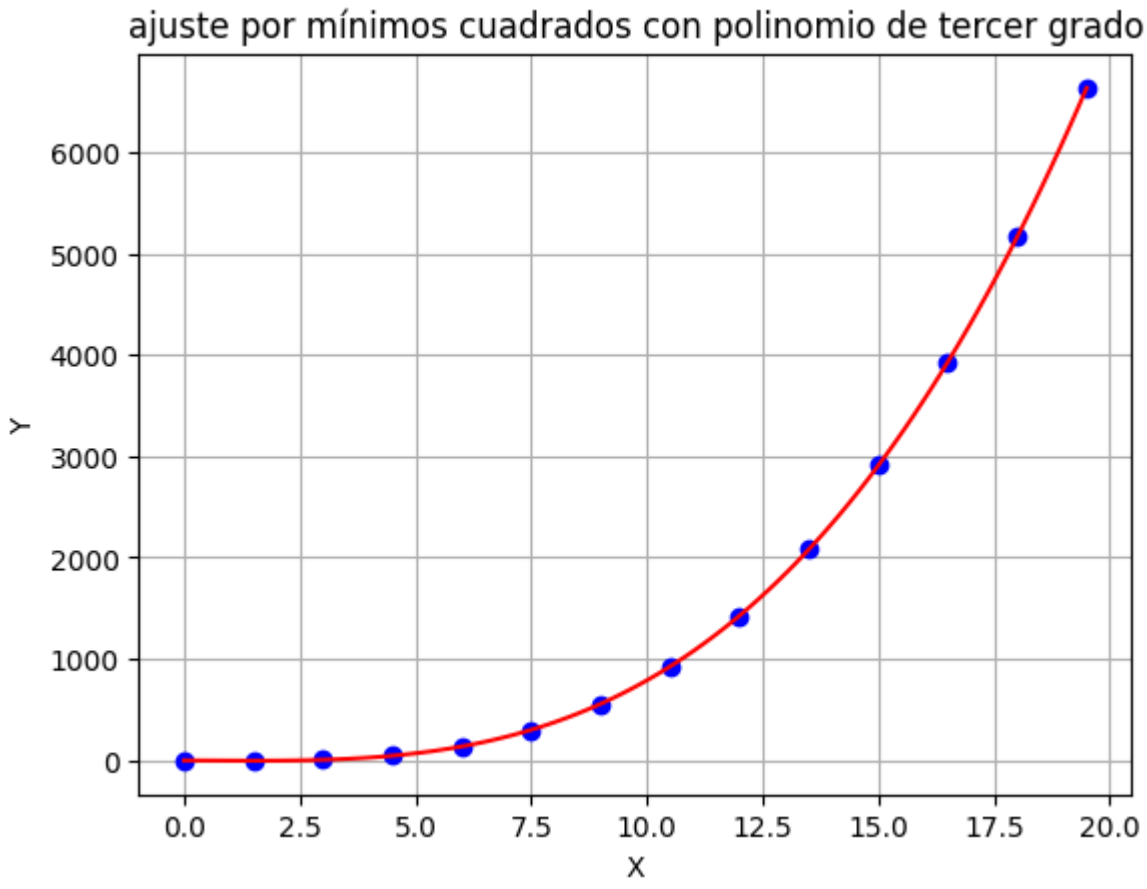
$$0$$

3.2.4 Ajuste por mínimos cuadrados (polinomio de tercer grado)

```
coe10, pol10 = data2.polinomio_min_cuad(3)
print(pol10)
```

```
        3     2
1 x - 2 x - 1 x - 1
```

```
data2.graficar_pol_np(pol10, 'ajuste por mínimos cuadrados con polinomio de tercer grado')
```




```
error10 = data2.error(y2, pol10(x2))
error10
```

```
1.4587791030841335e-22
```

3.3 Ejercicio práctico 2

Tabla 2. data ejercicio práctico 2

X	Y
1	0.500
2	0.143
3	0.071
4	0.044
5	0.029
6	0.021
7	0.016
8	0.013
9	0.010
10	0.008
11	0.007
12	0.006

```
#data ejercicio practico
x3 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
y3 = np.array([0.500, 0.143, 0.071, 0.044, 0.029, 0.021, 0.016, 0.013, 0.010, 0.008, 0.007, 0.006])
data3 = data(x3, y3)
x = sp.symbols('x')
```

3.3.1 Modelo 1

$$y = \frac{1}{ax^2 + bx + c}$$

Linealizando

$$\frac{1}{y} = ax^2 + bx + c$$
$$y_0 = ax^2 + bx + c$$

Entonces

X	$Y_0 = \frac{1}{Y}$
1	2.000
2	6.993
3	14.085
4	22.727
5	34.483
6	47.619
7	62.500
8	76.923
9	100.000
10	125.000
11	142.857
12	166.667

```
y0 = 1/y3
y0
```

```
array([  2.        ,   6.99300699,  14.08450704,  22.72727273,
        34.48275862,  47.61904762,  62.5       ,  76.92307692,
       100.        , 125.        , 142.85714286, 166.66666667])
```

```
data4 = data(x3,y0)
coe11, pol11 = data4.polinomio_min_cuad(2)
print(pol11)
```

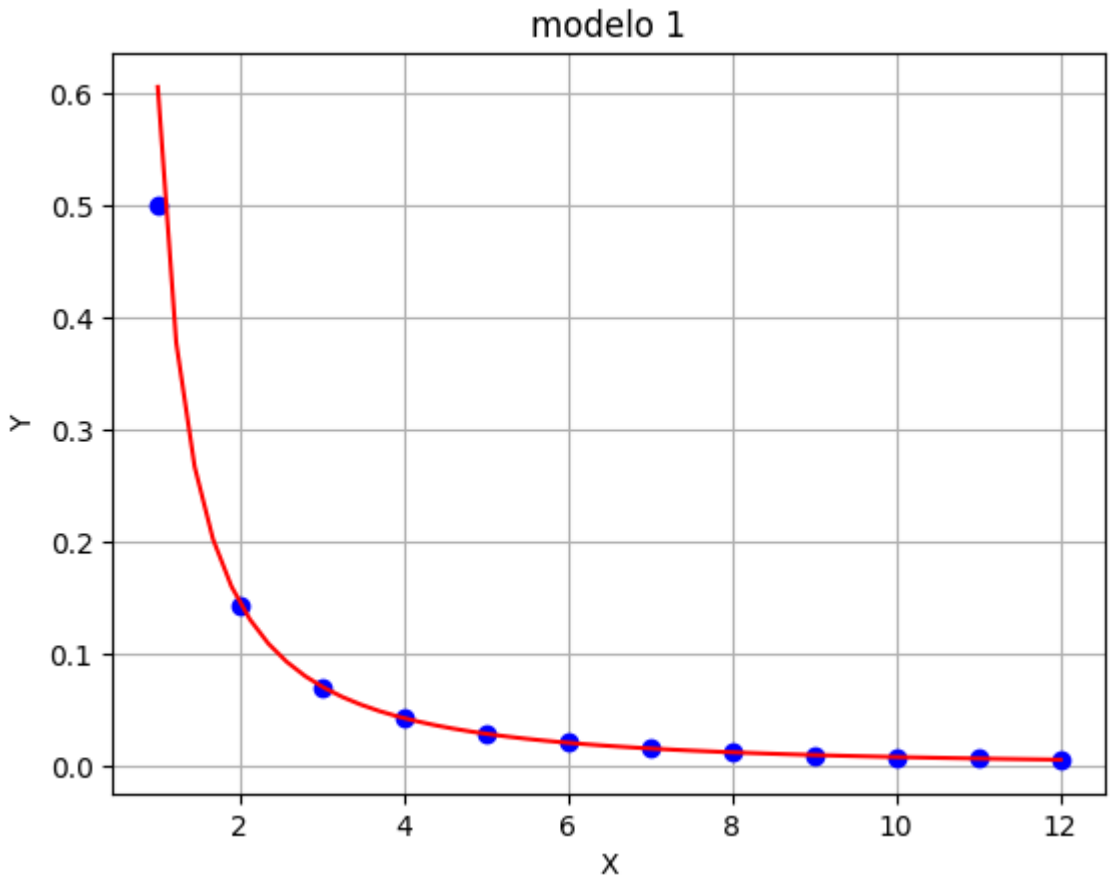
```
            2
0.9967 x + 2.215 x - 1.562
```

a = 0.9967, b = 2.215, c = - 1.562

Entonces:

$$y = \frac{1}{0.9967x^2 + 2.215x + -1.562}$$

```
f1 = 1 / (coe11[0]*(x**2) + coe11[1]*x + coe11[2])
data3.graficar_pol_sp(f1, 'modelo 1')
f1
```



$$\frac{1}{0.996679666873919x^2 + 2.21478863296958x - 1.56181811587646}$$

```
errorm1 = data3.error(y3, (f1.subs(x,i) for i in x3))
errorm1
```

$$0.0112858272437337$$

3.3.2 Modelo 2

$$y = ae^{bx}$$

Linealizando

$$Ln(y) = Ln(a) + bx$$
$$y_0 = A + bx$$

Entonces

X	$Y_0 = Ln(Y)$
1	-0.693
2	-1.945
3	-2.645
4	-3.124
5	-3.540
6	-3.863
7	-4.135
8	-4.343
9	-4.605
10	-4.828
11	-4.962
12	-5.116

```
data4.y = np.log(data3.y)
data4.y
```

```
array([-0.69314718, -1.94491065, -2.6450754 , -3.12356565, -3.54045945,
       -3.86323284, -4.13516656, -4.34280592, -4.60517019, -4.82831374,
       -4.96184513, -5.11599581])
```

```
coe12, pol12 = data4.polinomio_min_cuad(1)
print(pol12)
```

```
-0.3538 x - 1.351
```

A = -1.351, b = -0.3538

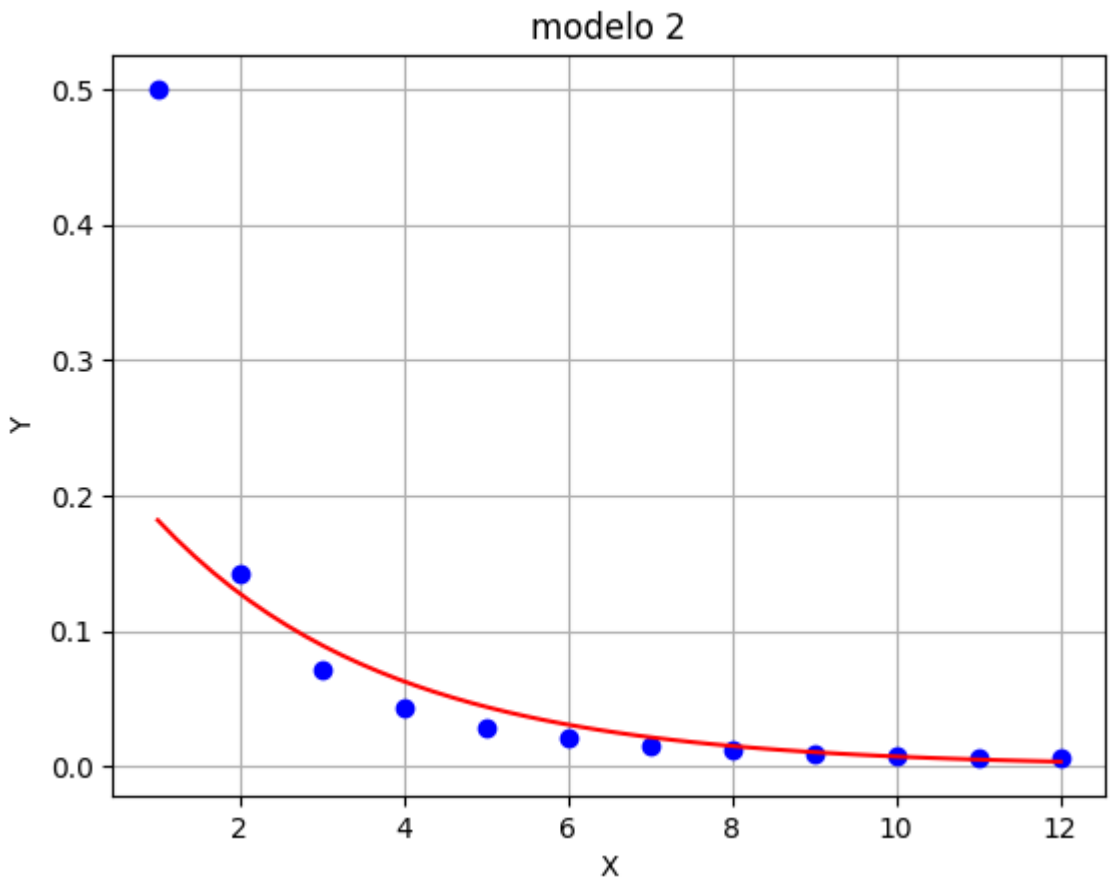
$$A = Ln(a) \rightarrow a = e^A$$

a = 0.2590

Entonces:

$$y = 0.259e^{-0.3538x}$$

```
f2 = ((sp.E)**coe12[1])*((sp.E)**(coe12[0]*x))
data3.graficar_pol_sp(f2, 'modelo 2')
f2
```



$$0.259090366355655e^{-0.353753179850303x}$$

```
errorm2 = data3.error(y3, (f2.subs(x,i) for i in x3))
errorm2
```

$$0.102510694288111$$

3.3.3 Modelo 3

$$y = ax^{-2b}$$

Linealizando

$$Ln(y) = Ln(a) - 2bLn(x)$$
$$y_0 = A + Bx_0$$

Entonces

$X_0 = Ln(X)$	$Y_0 = Ln(Y)$
0.000	-0.693
0.693	-1.945
1.099	-2.645
1.386	-3.124
1.609	-3.540
1.792	-3.863
1.946	-4.135
2.079	-4.343
2.197	-4.605
2.303	-4.828
2.398	-4.962
2.485	-5.116

```
data4.x = np.log(data3.x)
data4.x
```

```
array([0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791,
       1.79175947, 1.94591015, 2.07944154, 2.19722458, 2.30258509,
       2.39789527, 2.48490665])
```

```
coe13, pol13 = data4.polinomio_min_cuad(1)
print(pol13)
```

```
-1.778 x - 0.6879
```

A = -0.6879, B = -1.778

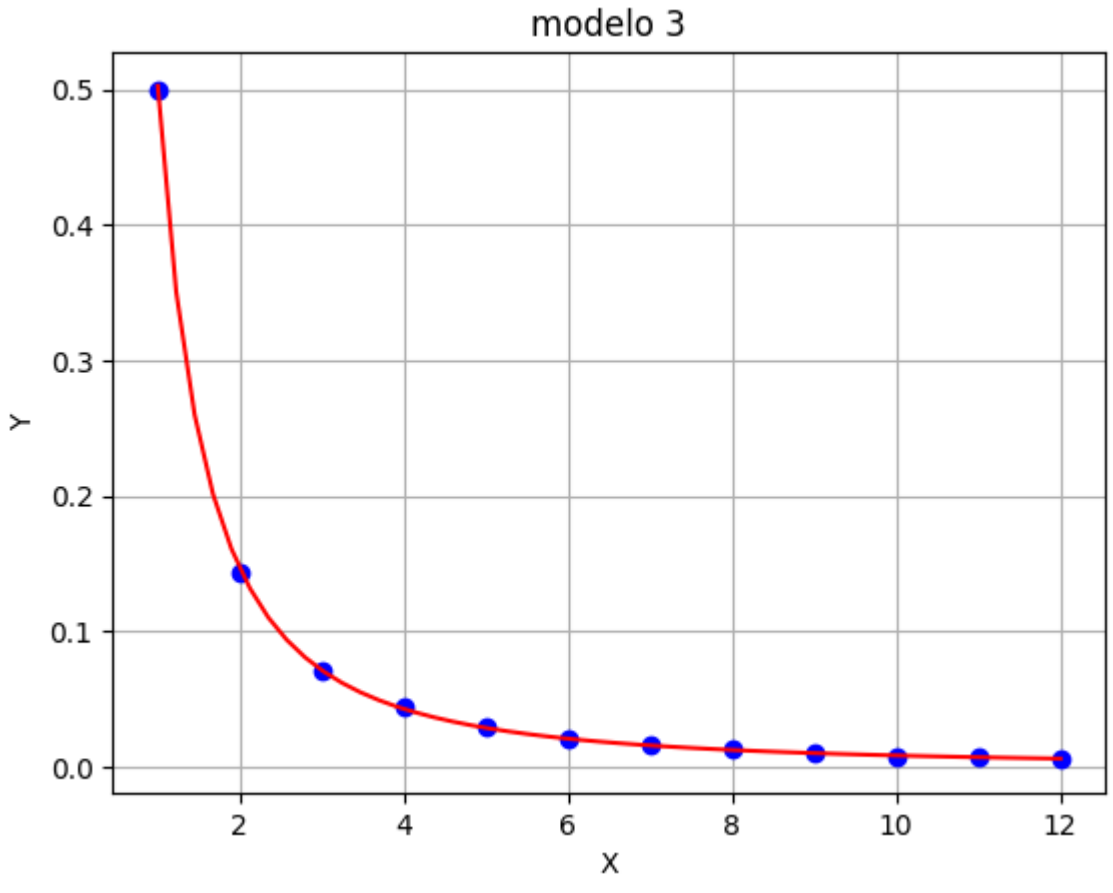
$$A = Ln(a) \rightarrow a = e^{A} = e^{-0.6879} = 0.503$$
$$B = -2b \rightarrow b = -\frac{B}{2} = -\frac{-1.778}{2} = 0.889$$

a = 0.503, b = 0.889

Entonces:

$$y = 0.503x^{-2(0.889)}$$

```
f3 = (sp.E**coe13[1])*(x**coe13[0])
data3.graficar_pol_sp(f3, 'modelo 3')
f3
```



$$\frac{0.502641813490475}{x^{1.7783948313526}}$$

```
errorm3 = data3.error(y3, (f3.subs(x,i) for i in x3))
errorm3
```

$$2.17463577041104 \cdot 10^{-5}$$

Tabla errores de modelos

modelo	1	2	3
error	0.0112858272437337	0.102510694288111	$2.1746357704114 \cdot 10^{-5}$

4. Análisis de resultados

Se analizará la efectividad de cada método presentado anteriormente a través de los distintos errores cuadráticos obtenidos. Se utilizó como base este tipo de error debido a que permite la diferenciabilidad, una ventaja considerable para estudios a profundidad en un futuro.

En el ejercicio práctico 1 se puede establecer que el polinomio interpolante de Newton, con un error de 0, y el ajuste por mínimos cuadrados utilizando un polinomio de grado 3, con un error de 1.200334220921008e-22, tienen una precisión mayor en comparación con la interpolación por método de Lagrange y coeficientes indeterminados, los cuales presentan un error relativamente menor, incluso se podría despreciar dependiendo del sistema que se modele. Cabe destacar que no es sorprendente que el polinomio de Newton tenga una mayor precisión, ya que analiza punto a punto las diferencias divididas, lo que permite reducir significativamente el error. Por otra parte, se debe indicar que existe cierta incertidumbre en cuanto a la precisión del método de los mínimos cuadrados en caso de ser estudiado con un polinomio de grado distinto de tres, que es el que mejor se adapta a los datos presentados.

En la última tabla del ejercicio práctico 2 se pueden observar los errores de cada modelo, e indica claramente que el modelo N°3 es el que más se aproxima a los datos dados, debido a que posee el error más bajo. En cambio, el modelo N°2 es el menos apropiado para

los datos dados, ya que su error es el más alto.

5. Conclusión

Gracias a los métodos numéricos, es posible estudiar de manera más precisa el comportamiento de un sistema. Esto nos permite conocer diversas formas de construir polinomios interpolantes que expresen los datos de manera más fiel. Además, nos brinda la oportunidad de estudiar diferentes casos, incluyendo casos extremos, y llevar un registro de la actividad del sistema. Es importante recordar que debemos utilizar el método que mejor se adapte al fenómeno o sistema que deseamos estudiar, analizar u optimizar, de acuerdo a los métodos previamente explicados.

6. Referencias

[1] **Analisis Numérico**, *Richard L. Burden • Douglas J. Faires • Annette M. Burden*, 10ma edición

[2] **Métodos Numéricos con Python**, *Ovalle D., Bernal M., Posada J.*, Editorial Politecnico Grancolombiano, (2021)

[3] **Lectura: Transformadas Integrales** *prof. Gilberto Noguera, Universidadd Central de Venezuela, Facultad de Ingeniería, Escuela de ingeniería Elétrica, Departamento de Eléctronica, Computación y Control* (2023)