Examen

Escuela Politécnica Nacional # Corrección examen 1

1. Los primeros tres términos diferentes a cero de la serie de Maclaurin para la función arcotangente son:

$x - \frac{1}{3} \cdot x^3 + \frac{1}{5} \cdot x^5$.

Calcule el error relativo en las siguientes aproximaciones de π mediante el polinomio (en lugar del arcotangente).

Asuma que $\pi = 3.14159$.

Aproximando con la Serie de Maclaurin de arctan(x)

Aproximación de arctan(x):

```
* Se sustituyen los valores seleccionados de x en la serie de Maclaurin y se calculan los pr

* **Ejemplo:**



  arctan(1/2)   1/2 - (1/3)(1/2)^3 + (1/5)(1/2)^5
```

Utilización de fórmulas trigonométricas:

```
* Se emplean fórmulas que relacionan  con funciones trigonométricas inversas (como arctan)
anterior.
```

- **Ejemplo:** La fórmula 4 * (arctan(1/2) + arctan(1/3)) se utiliza para aproximar .

Cálculo de la aproximación de :

- Se evalúan las expresiones obtenidas en el paso anterior para obtener un valor numérico aproximado de .

Cálculo del error:

```
* Se calcula la diferencia entre el valor aproximado y el valor real de  (3.14159).

* Se calcula el error relativo dividiendo el error absoluto entre el valor real.
```

1. Aproximación: $4 \cdot (\arctan(1/2) + \arctan(1/3))$

Redondee a 4 cifras significativas únicamente en la respuesta final de sus cálculos.

$\epsilon = 0.001269$.

¿En qué orden de magnitud está este error? Es decir, $\epsilon < 10^n\$$, $\$n = -2$.

2. Aproximación: $16 \cdot \arctan(1/5) - 4 \cdot \arctan(1/239)$

$\epsilon = 0.00009877$.

¿En qué orden de magnitud está este error? Es decir, $\epsilon < 10^n\$$, $\$n = -5$.

```python
import math

def arctan_approx(x):
    return x - (x**3)/3 + (x**5)/5

def calcular_pi(formula):
    return formula()

def error_relativo(valor_aproximado, valor_exacto):
    return abs((valor_aproximado - valor_exacto) / valor_exacto)

def orden_magnitud(error):
    return math.floor(math.log10(abs(error))) if error != 0 else -math.inf

pi_exacto = 3.14159

formula1 = lambda: 4 * (arctan_approx(1/2) + arctan_approx(1/3))
formula2 = lambda: 16 * arctan_approx(1/5) - 4 * arctan_approx(1/239)
```

```
pi_aprox1 = calcular_pi(formula1)
error_rel_1 = error_relativo(pi_aprox1, pi_exacto)
orden_magnitud_1 = orden_magnitud(error_rel_1)

pi_aprox2 = calcular_pi(formula2)
error_rel_2 = error_relativo(pi_aprox2, pi_exacto)
orden_magnitud_2 = orden_magnitud(error_rel_2)

print(f"Aproximación 1: {pi_aprox1:.6f}")
print(f"Error relativo 1: {error_rel_1:.4e}")
print(f"Orden de magnitud del error 1: {orden_magnitud_1}")

print(f"Aproximación 2: {pi_aprox2:.6f}")
print(f"Error relativo 2: {error_rel_2:.4e}")
print(f"Orden de magnitud del error 2: {orden_magnitud_2}")
```

```
Aproximación 1: 3.145576
Error relativo 1: 1.2688e-03
Orden de magnitud del error 1: -3
Aproximación 2: 3.141621
Error relativo 2: 9.8769e-06
Orden de magnitud del error 2: -6
```

2. Suponga que dos puntos (x_0, y_0) y (x_1, y_1) se encuentran en línea recta con $y_1 \neq y_0$.

Existen dos fórmulas para encontrar la intersección x de la línea:

Método A: $x = \frac{x_0 y_1 - x_1 y_0}{y_1 - y_0}$

Método B: $x = x_0 - \frac{(x_1 - x_0) y_0}{y_1 - y_0}$

Usando los datos $(x_0, y_0) = (1.31, 3.24)$ y $(x_1, y_1) = (1.93, 4.76)$, determine el valor real de la intersección x (asumiendo redondeo a 6 cifras significativas):

Cálculo con Aritmética de 3 Cifras Significativas

Método A:

2. Sustitución: x = (1.31 * 4.76 - 1.93 * 3.24) / (4.76 - 3.24)
3. Cálculo: x -0.00658

Método B:

2. Sustitución: x = 1.31 - ((1.93 - 1.31) * 3.24) / (4.76 - 3.24)
3. Cálculo: x -0.01

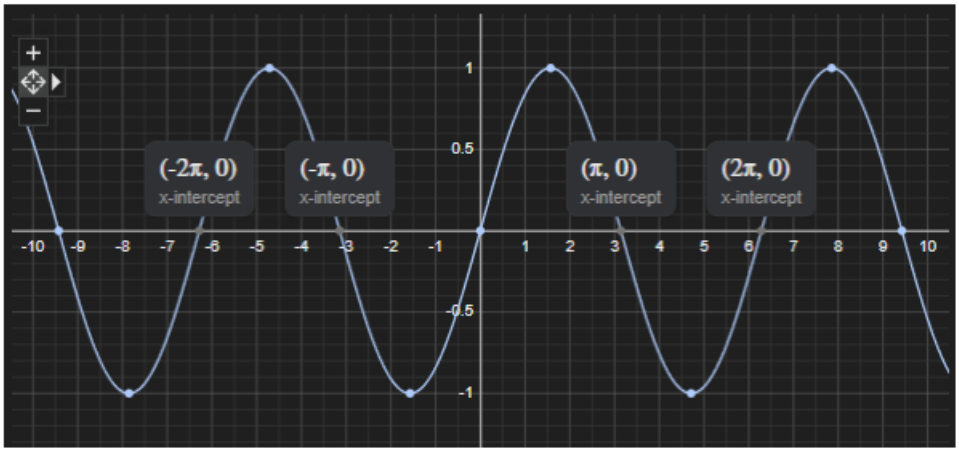
Cálculo del Error Relativo

- **Fórmula:** Error relativo = |(valor aproximado - valor exacto) / valor exacto|
- **Método A:** Error relativo 0.432
- **Método B:** Error relativo 0.136

Menor error

el método con menor error es el método B debido a que contiene una menor cantidad de multiplicaciones que son las operaciones que causan mayor error.

3. La función $sen(x)$ tiene infinitas soluciones



Este ejercicio se puede resolver al entender el método de la bisección, este métdo funciona al obtener el punto medio entre dos imágenes de signo distinto, realizando este proceso se pueden obtener las siguientes respuestas:

- $a = -2.5, b = -1$ Error las immágenes de estos puntos tienen el mismo signo.
- $a = -3.5, b - 3$ Error las immágenes de estos puntos tienen el mismo signo.
- $a = -4, b = 5$ pi es el valor que se encuentra más cercano al punto medio.
- $a = -5, b = 4$ -pi es el valor que se encuentra más cercano al punto medio.
- $a = -1, b = 5$ 0 es el valor que se encuentra más cercano al puntos medio.

- $a = 3, b = 5$ pi es el valor que se encuentra más cercano al punto medio.

4. El método de Newton para encontrar raíces se basa en la siguiente ecuación:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Cuál es la raíz de la ecuación:

$$x^3 + x = 1 + 3x^2$$

$$x_{sol} = 2.76929235$$

Qué sucede cuando:

- $x_0 = 3 \rightarrow x_{sol}$
- $x_0 = 1 \rightarrow$ Error [diverge u oscila]
- $x_0 = 0 \rightarrow$ Error [diverge u oscila]
- $x_0 = 1 + \frac{\sqrt{6}}{3} \rightarrow$ Error [división para 0]

```
import sympy as sp

# Definimos una variable simbólica x y la ecuación a resolver
x = sp.Symbol('x', real=True)  # Creamos una variable simbólica x que representa un número re
f = x**3 + x - 1 - 3*x**2  # Definimos la ecuación que queremos resolver

# Intentamos encontrar las soluciones para diferentes valores iniciales
try:
    # Iniciamos la búsqueda desde x=3
    sol = sp.nsolve(f, x, 3)  # Usamos nsolve para encontrar una solución numérica
    print("Solución encontrada (x0=3):", sol)
except Exception as e:  # Si ocurre un error, lo imprimimos
    print(e)

# Repetimos el proceso para otros valores iniciales
try:
    sol_1 = sp.nsolve(f, x, 1)
    print("Solución encontrada (x0=1):", sol_1)
```

```
except Exception as e:
    print(e)

try:
    sol_0 = sp.nsolve(f, x, 0)
    print("Solución encontrada (x0=0):", sol_0)
except Exception as e:
    print(e)

# Valor inicial especial
x0_special = 1 + sp.sqrt(6)/3
try:
    sol_special = sp.nsolve(f, x, x0_special)
    print("Solución encontrada (x0=1+sqrt(6)/3):", sol_special)
except Exception as e:
    print(e)
```

```
Solución encontrada (x0=3): 2.76929235423863
Solución encontrada (x0=1): 2.76929235423863
Could not find root within given tolerance. (1.94404735267969064602 > 2.16840434497100886801e
Try another starting point or tweak arguments.
cannot create mpf from sqrt(6)/3 + 1
```

5. El método de la secante se basa en la siguiente fórmula:

$$x_n = x_{n-1} - \frac{y_{n-1}(x_{n-1} - x_{n-2})}{y_{n-1} - y_{n-2}}$$

En base a esta fórmula, se ha generado el sigui

```
def secant_method(f, x0, x1, tol=1e-6, max_iter=100):
    x_prev = x0
    x_curr = x1
    iter_count = 0

    # Calcular f(x0) y f(x1) una vez
    f_prev = f(x_prev)
    f_curr = f(x_curr)

    while abs(f_curr) > tol and iter_count < max_iter:
        # Calcular el siguiente valor usando la fórmula del método de la secante
        x_next = x_curr - f_curr * (x_curr - x_prev) / (f_curr - f_prev)

        # Verificar si el cambio en x es menor que la tolerancia
```

```
        if abs(x_next - x_curr) < tol:
            break

        # Actualizar los valores para la siguiente iteración
        x_prev = x_curr
        x_curr = x_next
        f_prev = f_curr
        f_curr = f(x_curr)
        iter_count += 1

    return x_curr, iter_count
```

Ejemplo 1

```
i = 0


def func(x):
    global i
    i += 1
    y = x**3 - 3 * x**2 + x - 1
    print(f"Llamada i={i}\t x={x:.5f}\t y={y:.2f}")
    return y


secant_method(func, x0=2, x1=3)
```

```
Llamada i=1  x=2.00000   y=-3.00
Llamada i=2  x=3.00000   y=2.00
Llamada i=3  x=2.60000   y=-1.10
Llamada i=4  x=2.74227   y=-0.20
Llamada i=5  x=2.77296   y=0.03
Llamada i=6  x=2.76922   y=-0.00
Llamada i=7  x=2.76929   y=-0.00

(2.7692921651959503, 5)
```

Ejemplo2

```
i = 0
import math


def func(x):
    global i
    i += 1
    y = math.sin(x) + 0.5
    print(f"Llamada i={i}\t x={x:.5f}\t y={y:.2f}")
    return y


secant_method(func, x0=2, x1=3)
```

```
Llamada i=1  x=2.00000   y=1.41
Llamada i=2  x=3.00000   y=0.64
Llamada i=3  x=3.83460   y=-0.14
Llamada i=4  x=3.68602   y=-0.02
Llamada i=5  x=3.66399   y=0.00
Llamada i=6  x=3.66520   y=-0.00
Llamada i=7  x=3.66519   y=-0.00

(3.66519143172732, 5)
```

Luego de optimizar el código y utilizando (x_0=2, x_1=3), conteste: ¿Cuál es el número mínimo de llamadas a la función para llegar a la raíz en el Ejemplo 1? * i= 8 ¿Cuál es el número mínimo de llamadas a la función para llegar a la raíz en el Ejemplo 2? * i=7

7. La interpolación de Lagrange nos permite construir un polinomio que pasa exactamente por un conjunto de puntos dados. Formalmente, el polinomio de Lagrange $P(x)$ se define como:

$$P(x) = \sum_{k=0}^{n} y_k \prod_{i=0, i \neq k}^{n} \frac{x - x_i}{x_k - x_i}$$

donde (x_k, y_k) son los puntos de datos.

Consideremos los puntos (0, 0), (1, 1), (2, 2) y (3, 3). Intuitivamente, la relación entre las coordenadas x e y es una función lineal. Por tanto, el polinomio de Lagrange de menor grado que interpola estos puntos es simplemente:

$$P(x) = x$$

Evaluando en x = 3.78 y x = 19.102, obtenemos:

$$P(3.78) = 3.78$$
$$P(19.102) = 19.102$$

El polinomio simplificado tiene orden =1.

```
import sympy as sp

x = sp.Symbol('x')
puntos = [(0, 0), (1, 1), (2, 2), (3, 3)]

def lagrange_basis(puntos, k, x):
    xk, _ = puntos[k]
    Lk = 1
    for i, (xi, _) in enumerate(puntos):
        if i != k:
            Lk *= (x - xi) / (xk - xi)
    return Lk

P = sum(yk * lagrange_basis(puntos, k, x) for k, (xk, yk) in enumerate(puntos))

P_simplificado = sp.simplify(P)

val_378 = P_simplificado.subs(x, 3.78)
val_19102 = P_simplificado.subs(x, 19.102)

print(P_simplificado)
print(val_378)
print(val_19102)
```

```
x
3.78000000000000
19.1020000000000
```

7.

$$P(x) = \sum_{i=0}^{n} L_i(x) y_i$$

$$L_k(x) = \prod_{i=0, i \neq k}^{n} \frac{x - x_i}{x_k - x_i}$$

Considerando los puntos: $(0,0), (1,1), (2,2), (3,3)$

$$P(x) = L_0(x)y_0 + L_1(x)y_1 + L_2(x)y_2 + L_3(x)y_3$$
$$P(x) = L_0(x)(0) + L_1(x)(1) + L_2(x)(2) + L_3(x)(3)$$
$$P(x) = L_1(x) + 2L_2(x) + 3L_3(x)$$

$$L_1(x) = \frac{(x - x_0)(x - x_2)(x - x_3)}{(x_1 - x_0)(x_1 - x_2)(x_1 - x_3)} = \frac{x(x-2)(x-3)}{2}$$
$$L_2(x) = \frac{(x - x_0)(x - x_1)(x - x_3)}{(x_2 - x_0)(x_2 - x_1)(x_2 - x_3)} = \frac{x(x-1)(x-3)}{-2}$$
$$L_3(x) = \frac{(x - x_0)(x - x_1)(x - x_2)}{(x_3 - x_0)(x_3 - x_1)(x_3 - x_2)} = \frac{x(x-1)(x-2)}{6}$$

Sustituyendo:

$$P(x) = \frac{x(x-2)(x-3)}{2} + 2\left(\frac{x(x-1)(x-3)}{-2}\right) + 3\left(\frac{x(x-1)(x-2)}{6}\right)$$

$$P(x) = \frac{x^3 - 5x^2 + 6x}{2} - \frac{2x^3 - 8x^2 + 6x}{2} + \frac{x^3 - 3x^2 + 2x}{2}$$

$$P(x) = \frac{2x}{2}$$

$$P(x) = x$$

8. Dados los puntos (-1,1),(0,5),(1,3), se ha obtenido los splines cúbicos correspondientes. Sin embargo, al observar la figura, usted no se siente satisfecho con la pendiente resultante en el punto (x_1,y_1). Y decide intentar una modificación a las ecuaciones, tal que los splines sean tangentes a una pendiente deseada m en el punto (x_1,y_1)

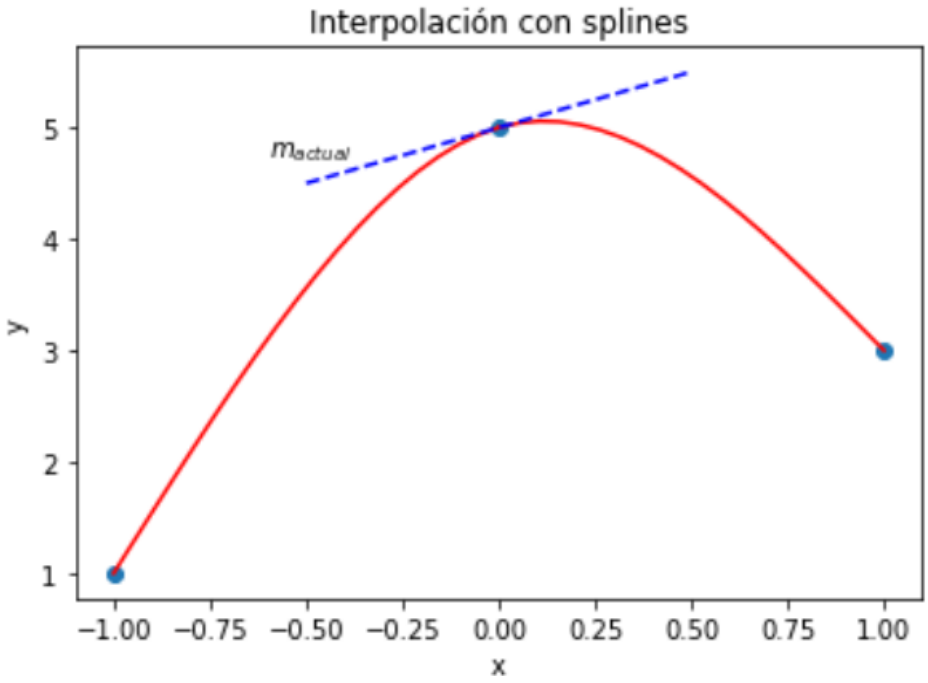


Figure 1: pregunta8

¿En caso de ser posible, y bajo qué condiciones se puede encontrar los splines cúbicos que cumplan con la condición de m?

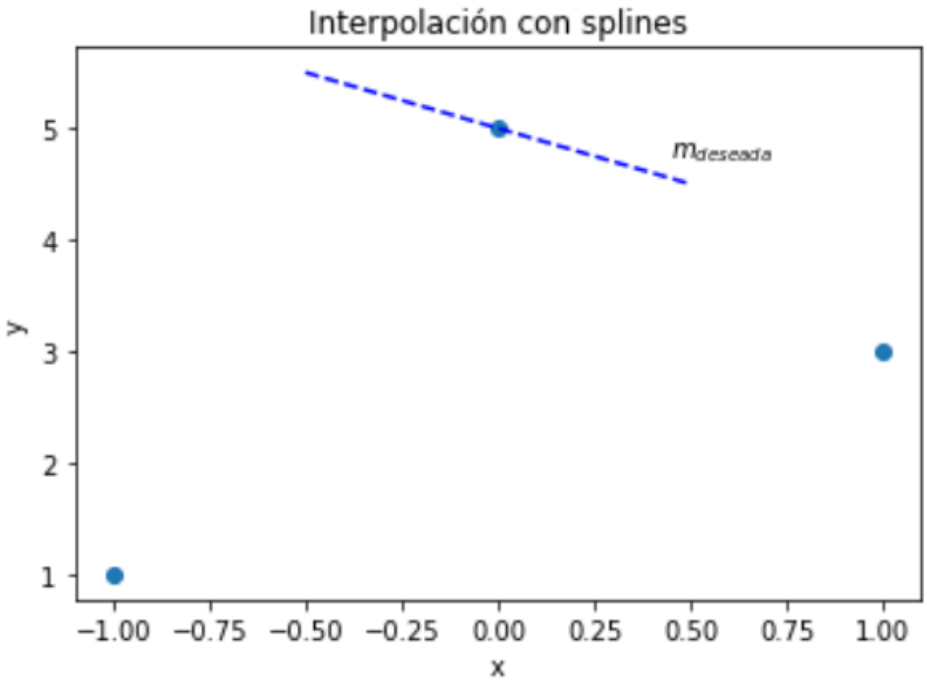


Figure 2: pregunta8.1

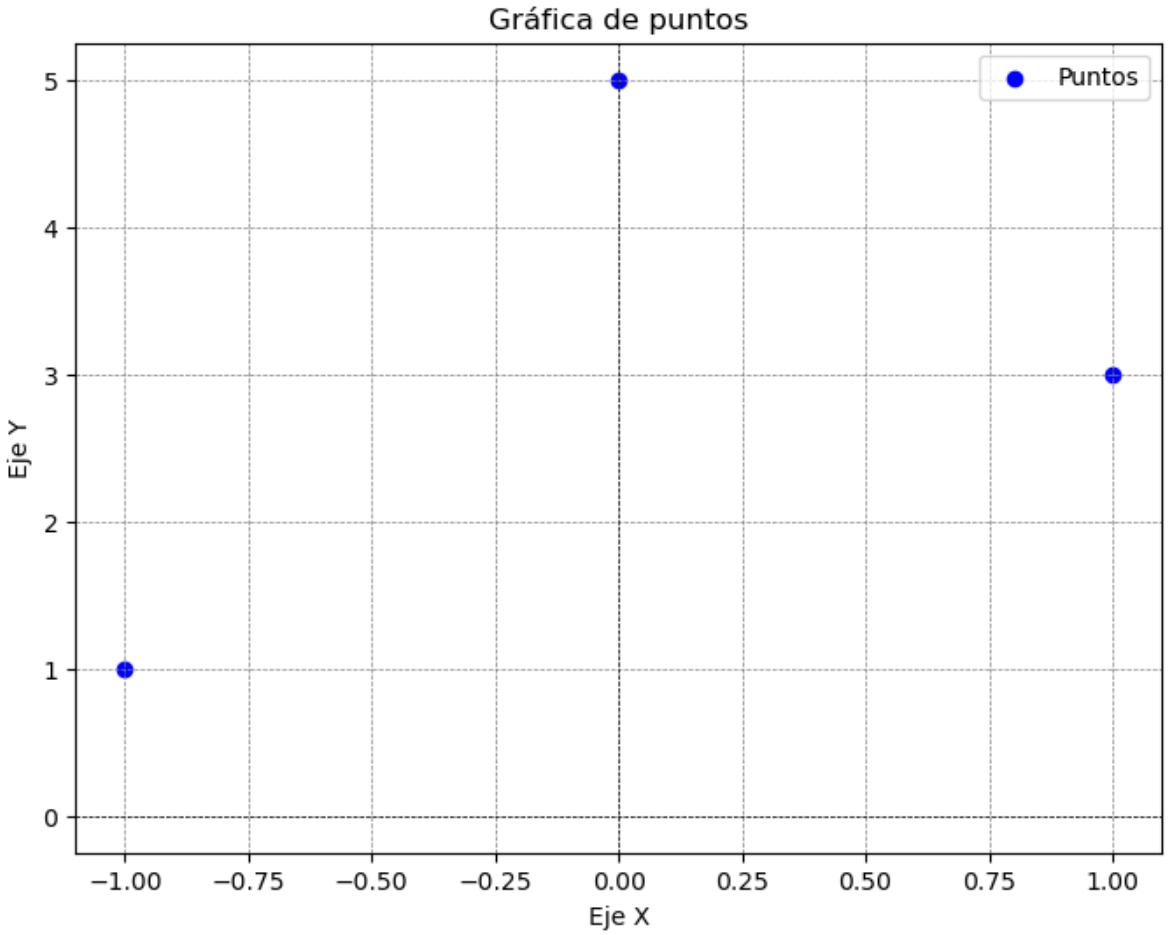
Determine la ecuación que se debe modificar para poder cumplir con el requisito de m. * $S_0'(x) = S_1'(x)$ * $S_0''(x_0) = 0$ * $S_1(x_2) = y_2$ * $S_0(x_0) = y_0$ * $S_1''(x_2) = 0$ * Ninguna de las anteriores * $S_0''(x_1) = S_1''(x_1)$ * Sin solución única.Depende de los valores de m * $S_1(x_1) = y_1$ * $S_0 = (x_1) = y_1$ Respuesta: La ecuación que se debe cambiar es la $S_0''(x_1) = S_1''(x_1)$ debido a que debemos cumplir con la condición de la pendiente, se verá remplazada por la ecuación $S_0'(0) = -2$ y $S_1'(0) = -2$

Escriba la expresión del spline S_0. En caso de no existir solución, llene los casilleros con Nan

Para realizar este ejercicio, primero realizaremos una gráfica de los puntos

```
import matplotlib.pyplot as plt
x = [-1, 0, 1]
y = [1, 5, 3]
plt.figure(figsize=(8, 6))
plt.scatter(x, y, color='blue', label="Puntos")
plt.title("Gráfica de puntos")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
plt.axvline(0, color='black', linewidth=0.5, linestyle='--')
```

```
plt.grid(color='gray', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()
```



Planteamos las ecuaciones: ### $S_0(-1) = 1 * a_0 + b_0(-1-(-1)) + c_0(-1-(-1))^2 + d_0(-1-(-1))^3 = 1 * a_0 = 1$

$S_0(0) = 5$

- $a_0 + b_0(0-(-1)) + c_0(0-(-1))^2 + d_0(0-(-1))^3 = 5$
- $a_0 + b_0 + c_0 + d_0 = 5$

$S_0'(0) = -2$

- $b_0 + 2c_0(0+1) + 3d_0 = -2$

$S_0''(-1) = 0$

- $2c_0 + 6d_0(-1+1) = 0$
- $c_0 = 0$

$S_1(0) = 5$

- $a_1 + b_1(0-0) + c_1(0-0)^2 + d_1(0-0)^3$
- $a_1 = 5$

$S_1(1) = 3$

- $a_1 + b_1(1-0) + c_1(1-0)^2 + d_1(1-0)^3$
- $a_1 + b_1 + c_1 + d_1 = 3$

$S_1'(0) = -2$

- $b_1 + 2c_1(0-0) + 3d_1(0-0) = -2$

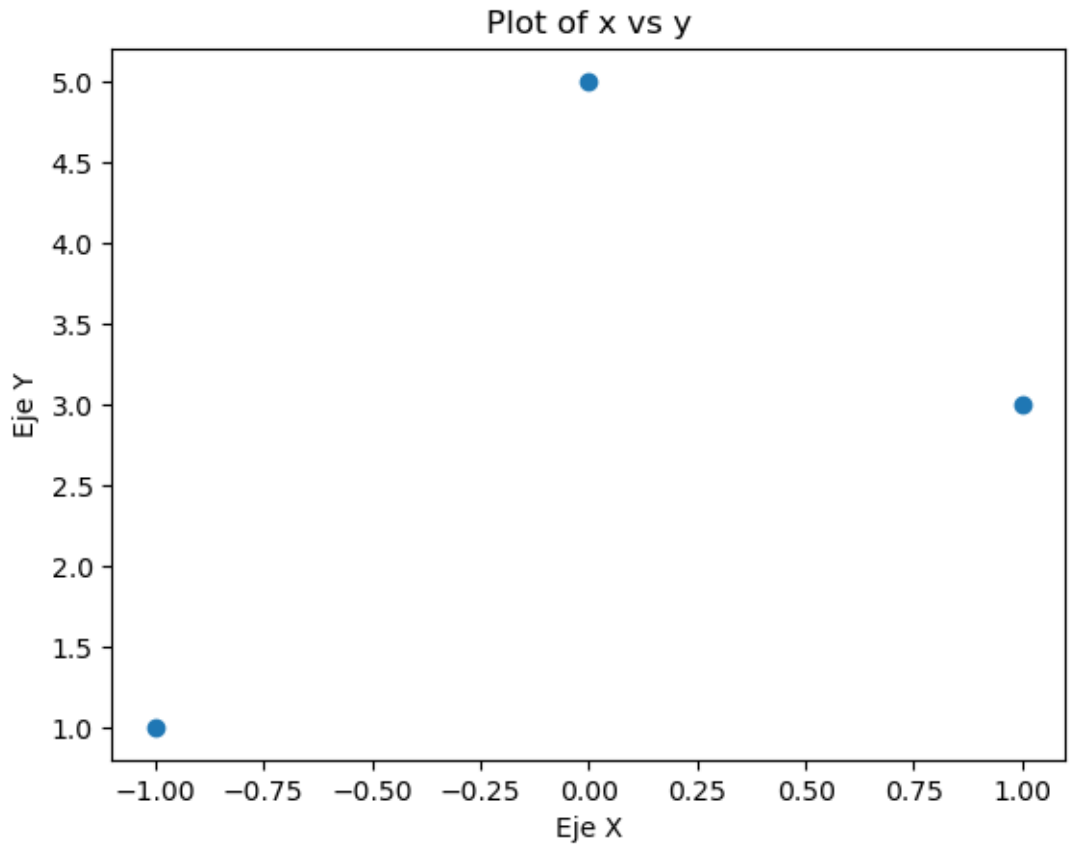
$S_1''(1) = 0$

- $2c_1 + 6d_1(1)0$
- $2c_1 + 6d_1 = 0$ ## Respuesta
- $S_0(x) = -3 * (x-1)^3) + 0 * (x-(-1))^2 + 7 * (x-1) + 1$
- $S_1(x) = 0 * (x-0)^3 + 0 * (x-0)^2 + -2 * ()$

```
xs = [-1,0,1]
ys = [1,5,3]

import matplotlib.pyplot as plt

plt.scatter(xs, ys)
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.title("Plot of x vs y")
plt.show()
```




```
def Spline(x: float, x0: float, pars: dict[int, float]) -> float:
    a = pars["a"]
    b = pars["b"]
    c = pars["c"]
    d = pars["d"]
    return a + b * (x - x0) + c * (x - x0) ** 2 + d * (x - x0) ** 3
```

```
import numpy as np

s = [
    {"a": 1, "b": 7, "c": 0, "d": -3},
    {"a": 5, "b": -2, "c":0, "d": 0},
]
for i, x_i in enumerate(xs[:-1]):
    _x = np.linspace(x_i, xs[i + 1], 20)
    y = Spline(_x, x_i, s[i])

    plt.plot(_x, y, color="pink")

plt.scatter(xs, ys)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Interpolación con splines")
plt.show()
```