

# Actividad extracurricular 08 - Corrección Examen

realizado por : Correa Adrian

fecha de entrega: 11/12/2024

link de Github: <https://github.com/afca2002/Actividad-extracurricular-08.git>

## Cálculo del Error Relativo en Aproximaciones de $\pi$ mediante Polinomios

### CASO 1

Calcule el error relativo en la siguiente aproximación de  $\pi$  mediante el polinomio de Maclaurin para el arcotangente. Redondee a 4 cifras significativas únicamente en la respuesta final de sus cálculos.

Aproximación:

$$4(\arctan(1/2) + \arctan(1/3))$$

### CASO 2

Calcule el error relativo en la siguiente aproximación de  $\pi$  mediante el polinomio de Maclaurin para el arcotangente. Redondee a 4 cifras significativas únicamente en la respuesta final de sus cálculos.

Aproximación:

$$16(\arctan(1/5)) - 4(\arctan(1/239))$$

Asuma que  $\pi = 3.14159$ .

---

## Desarrollo Matemático

### Fórmula del Error Relativo

El error relativo se calcula como:

$$\epsilon = \frac{\pi_{\text{real}} - \pi_{\text{aprox}}}{\pi_{\text{real}}}$$

donde:

- $\pi_{\text{real}} = 3.14159$

---

## Aproximación de $\pi$ usando la primera fórmula: $4(\arctan(1/2) + \arctan(1/3))$

### Calcular $\arctan(1/2)$ usando la serie de Maclaurin

$$\arctan(1/2) \approx 0.5 - \frac{1}{3}(0.5^3) + \frac{1}{5}(0.5^5)$$

$$\arctan(1/2) \approx 0.5 - 0.04167 + 0.00625$$

$$\arctan(1/2) \approx 0.46458$$

### Calcular $\arctan(1/3)$ usando la serie de Maclaurin

$$\arctan(1/3) \approx 0.33333 - \frac{1}{3}(0.33333^3) + \frac{1}{5}(0.33333^5)$$

$$\arctan(1/3) \approx 0.33333 - 0.01235 + 0.000823$$

$$\arctan(1/3) \approx 0.32180$$

### Sustituyendo los valores calculados

$$\pi_{\text{aprox}_1} \approx 4(0.46458 + 0.32180)$$

$$\pi_{\text{aprox}_1} \approx 4 \times 0.78638$$

$$\pi_{\text{aprox}_1} \approx 3.1456$$

### Calcular el error relativo para la primera aproximación

$$\epsilon_1 = \left| \frac{\pi_{\text{aprox}_1} - \pi_{\text{real}}}{\pi_{\text{real}}} \right|$$
$$\epsilon_1 \approx \left| \frac{3.1456 - 3.14159}{3.14159} \right|$$
$$\epsilon_1 \approx 0.0013$$

### Orden de magnitud del error

$$\epsilon_1 < 10^{-3}$$

## Aproximación de $\pi$ usando la segunda fórmula: $16 \arctan(1/5) - 4 \arctan(1/239)$

### Calcular $\arctan(1/5)$ usando la serie de Maclaurin

$$\arctan(1/5) \approx 0.2 - \frac{1}{3}(0.2^3) + \frac{1}{5}(0.2^5)$$

$$\arctan(1/5) \approx 0.2 - 0.002667 + 0.000064$$

$$\arctan(1/5) \approx 0.197397$$

## Calcular $\arctan(1/239)$ usando la serie de Maclaurin

$$\arctan(1/239) \approx 0.004184 - \frac{1}{3}(0.004184^3) + \frac{1}{5}(0.004184^5)$$

$$\arctan(1/239) \approx 0.004184 - 0.000000024 + 0$$

$$\arctan(1/239) \approx 0.004184$$

## Sustituyendo los valores calculados

$$\pi_{\text{aprox}_2} \approx 16 \times 0.197397 - 4 \times 0.004184$$

$$\pi_{\text{aprox}_2} \approx 3.158352 - 0.016736$$

$$\pi_{\text{aprox}_2} \approx 3.1416$$

## Calcular el error relativo para la segunda aproximación

$$\epsilon_2 = \left| \frac{\pi_{\text{aprox}_2} - \pi_{\text{real}}}{\pi_{\text{real}}} \right|$$

$$\epsilon_2 \approx \left| \frac{3.1416 - 3.14159}{3.14159} \right|$$

$$\epsilon_2 \approx 0.0$$

## Orden de magnitud del error

$$\epsilon_2 < 10^{-5}$$

```
In [ ]: # CASO 1
import math

# Definir la serie de Maclaurin
def maclaurin_arctan(x):
    term1 = x
    term2 = -(x**3) / 3
    term3 = (x**5) / 5
    return term1 + term2 + term3

arctan_1_2 = maclaurin_arctan(1/2)
arctan_1_3 = maclaurin_arctan(1/3)

pi_aprox = 4 * (arctan_1_2 + arctan_1_3)

# Valor verdadero de pi
pi_real = 3.14159

# Calcular el error relativo
error_relativo = abs((pi_aprox - pi_real) / pi_real)

# Determinar el orden de magnitud del error
orden_magnitud_error = math.floor(math.log10(error_relativo))

# Imprimir resultados
print(f"Error relativo: {error_relativo:.6f}")
print(f"El error está en el orden de magnitud: 10^{orden_magnitud_error}")
```

Error relativo: 0.001269

El error está en el orden de magnitud: 10^-3

```
In [ ]: # CASO 2
import math

# Definir la serie de MacLaurin
def maclaurin_arctan(x):
    term1 = x
    term2 = -(x**3) / 3
    term3 = (x**5) / 5
    return term1 + term2 + term3

arctan_1_5 = maclaurin_arctan(1/5)
arctan_1_239 = maclaurin_arctan(1/239)

pi_aprox_2 = 16 * arctan_1_5 - 4 * arctan_1_239

pi_real = 3.14159

# Calcular el error relativo
error_relativo_2 = abs((pi_aprox_2 - pi_real) / pi_real)

# Determinar el orden de magnitud del error
orden_magnitud_error_2 = math.floor(math.log10(error_relativo_2))

print(f"Error relativo: {error_relativo_2:.10f}")
print(f"El error está en el orden de magnitud: 10^{orden_magnitud_error_2}")


```

Error relativo: 0.0000098769  
 El error está en el orden de magnitud: 10^-6

## PREGUNTA 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:

- $(x_0, y_0) = (1.31, 3.24)$
  - $(x_1, y_1) = (1.93, 4.76)$
- 

## Desarrollo Matemático

## 1. Valor real de la intersección $x$ (redondeado a 6 cifras significativas)

Utilizando el **Método A** con aritmética exacta:

$$x_{\text{real}} = \frac{x_0 y_1 - x_1 y_0}{y_1 - y_0}$$

Sustituyendo los valores:

$$x_{\text{real}} = \frac{(1.31 \cdot 4.76) - (1.93 \cdot 3.24)}{4.76 - 3.24}$$

$$x_{\text{real}} = \frac{6.2396 - 6.2532}{1.52} = -0.0115789$$

## Redondeando a 6 cifras significativas:

### 2. Cálculo del valor de $x$ redondeado a 3 cifras significativas

#### Método A

Usando la fórmula:

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

Sustituyendo los valores y redondeando en cada operación a 3 cifras significativas:

$$x = \frac{(1.31 \cdot 4.76) - (1.93 \cdot 3.24)}{4.76 - 3.24} = \frac{6.24 - 6.25}{1.52} = -0.00658$$

$$\epsilon_A = \frac{-0.011579 - (-0.00658)}{-0.011579} = \frac{-0.011579 + 0.00658}{-0.011579} = \frac{-0.005}{-0.011579} = 0.432$$

Redondeando a 3 cifras significativas:

$$\epsilon_A = 0.432$$

#### Método B

Usando la fórmula:

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

Sustituyendo los valores y redondeando en cada operación a 3 cifras significativas:

$$x = 1.31 - \frac{(1.93 - 1.31) \cdot 3.24}{4.76 - 3.24}$$

$$x = 1.31 - \frac{0.62 \cdot 3.24}{1.52} = 1.31 - 1.32 = -0.01$$

$$\epsilon_B = \frac{-0.011579 - (-0.01)}{-0.011579} = \frac{-0.011579 + 0.01}{-0.011579} = \frac{-0.001579}{-0.011579} = 0.136$$

Redondeando a 3 cifras significativas:

$$\epsilon_B = 0.136$$

# Resultados

## Método A

- Intersección aproximada:  $x = -0.00658$
- Error relativo:  $\epsilon = 0.432$

## Método B

- Intersección aproximada:  $x = -0.01$
- Error relativo:  $\epsilon = 0.136$

## ¿Cuál método es mejor?

El **Método B** tiene un error relativo menor ( $\epsilon = 0.136$ ) en comparación con el **Método A** ( $\epsilon = 0.432$ ). Por lo tanto, **el Método B es mejor.**

```
In [ ]: import math

# Redondea un número a una cantidad específica de cifras significativas
def round_to_significant_figures(value, significant_figures):
    if value == 0:
        return 0
    else:
        return round(value, significant_figures - int(math.floor(math.log10(abs(
            value)))))

# Coordenadas de los dos puntos dados
x0, y0 = 1.31, 3.24
x1, y1 = 1.93, 4.76

# Valor real de la intersección (redondeado a 6 cifras significativas)
numerador_real = x0 * y1 - x1 * y0
denominador_real = y1 - y0
x_real = numerador_real / denominador_real
x_real = round_to_significant_figures(x_real, 6)

# Método A: Cálculo de la intersección aproximada y error relativo
numerador_A = round_to_significant_figures(x0 * y1, 3) - round_to_significant_fi
denominador_A = round_to_significant_figures(y1 - y0, 3)
x_A = round_to_significant_figures(numerador_A / denominador_A, 3)
epsilon_A = round_to_significant_figures((x_real - x_A) / x_real, 3)

# Método B: Cálculo de la intersección aproximada y error relativo
delta_x = round_to_significant_figures(x1 - x0, 3)
producto_B = round_to_significant_figures(delta_x * y0, 3)
division_B = round_to_significant_figures(producto_B / denominador_real, 3)
x_B = round_to_significant_figures(x0 - division_B, 3)
epsilon_B = round_to_significant_figures((x_real - x_B) / x_real, 3)

# Resultados
print("Resultados:")
print(f"Valor real de la intersección (6 cifras significativas): x_real = {x_rea
print(f"\nMétodo A:")
print(f"Intersección aproximada: x_A = {x_A}")
```

```

print(f"Error relativo: epsilon_A = {epsilon_A}")
print(f"\nMétodo B:")
print(f"Intersección aproximada: x_B = {x_B}")
print(f"Error relativo: epsilon_B = {epsilon_B}")

# Determinar cuál método es mejor
if abs(epsilon_A) < abs(epsilon_B):
    print("\nEl Método A es mejor.")
else:
    print("\nEl Método B es mejor.")

```

Resultados:

Valor real de la intersección (6 cifras significativas):  $x_{\text{real}} = -0.0115789$

Método A:

Intersección aproximada:  $x_A = -0.00658$

Error relativo:  $\text{epsilon}_A = 0.432$

Método B:

Intersección aproximada:  $x_B = -0.01$

Error relativo:  $\text{epsilon}_B = 0.136$

El Método B es mejor.

## PREGUNTA 3)

### Método de Bisección aplicado a $\sin(x)$

La función  $\sin(x)$  tiene infinitas soluciones  $\{\dots, -2\pi, -\pi, 0, \pi, 2\pi, \dots\}$ .

**¿A cuál solución converge el método de la Bisección en los siguientes intervalos?**

### Intervalos y Resultados:

1.  $a = -5, b = 4 \rightarrow$  Converge a  $-\pi$
2.  $a = -1, b = 2 \rightarrow$  Converge a 0
3.  $a = 3, b = 5 \rightarrow$  Converge a  $\pi$
4.  $a = -3.5, b = 3 \rightarrow$  Error [sin cambio de signo]
5.  $a = -4, b = 5 \rightarrow$  Converge a  $\pi$
6.  $a = -2.5, b = -1 \rightarrow$  Error [sin cambio de signo]

### Proceso Matemático:

El método de Bisección se aplica de la siguiente manera:

1. Verificar el cambio de signo:  $f(a) \cdot f(b) < 0$  Esto asegura que existe al menos una raíz en el intervalo.
2. Calcular el punto medio:  $c = \frac{a+b}{2}$
3. Evaluar el signo de  $f(c)$ :
  - Si  $f(a) \cdot f(c) < 0$ , la raíz está en  $[a, c]$ .
  - Si  $f(c) \cdot f(b) < 0$ , la raíz está en  $[c, b]$ .

4. Repetir iterativamente hasta que la longitud del intervalo sea menor que una tolerancia definida.

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

# Crear un rango de valores para x
x = np.linspace(-10, 10, 1000)

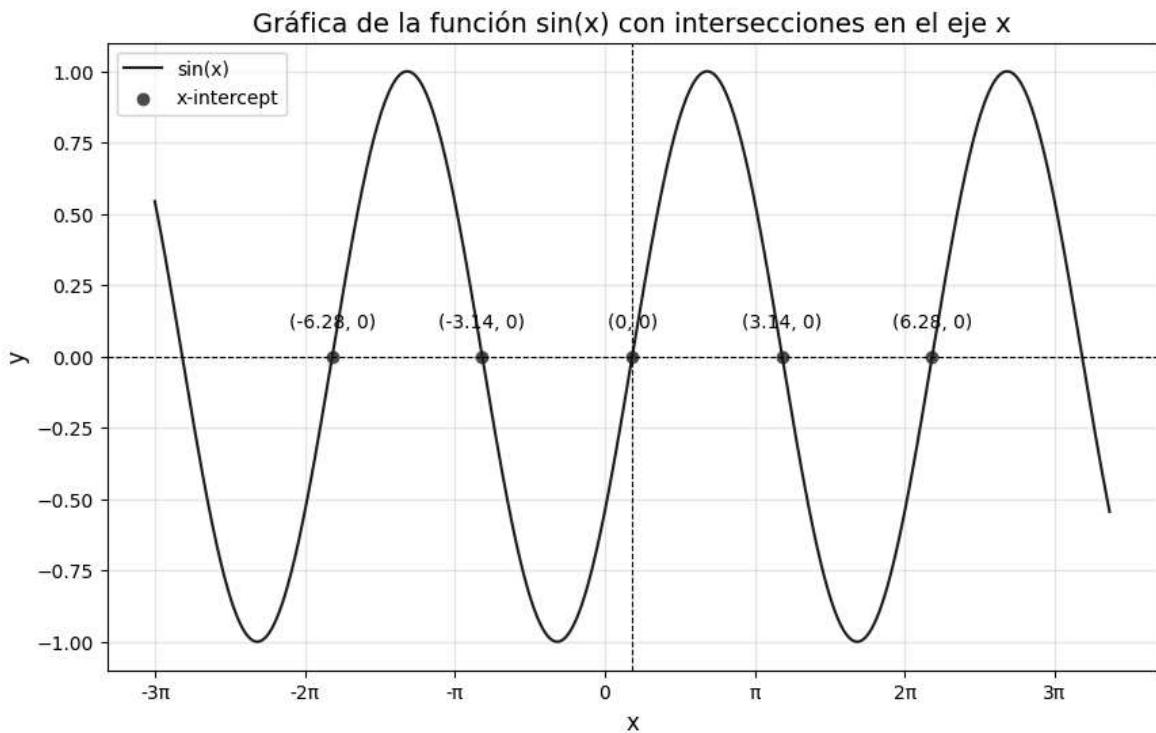
# Calcular el seno para cada valor de x
y = np.sin(x)

# Crear la gráfica
plt.figure(figsize=(10, 6))
plt.plot(x, y, label="sin(x)", color="blue")

# Agregar puntos importantes (intersecciones con el eje x)
x_intercepts = [-2 * np.pi, -np.pi, 0, np.pi, 2 * np.pi]
y_intercepts = [0] * len(x_intercepts)
plt.scatter(x_intercepts, y_intercepts, color="red", label="x-intercept")

# Etiquetas para las intersecciones
for x_i in x_intercepts:
    plt.text(x_i, 0.1, f"{{round(x_i, 2)}}, 0)", fontsize=10, color="black", ha=""

# Añadir líneas y mejorar estética
plt.axhline(0, color="black", linewidth=0.8, linestyle="--")
plt.axvline(0, color="black", linewidth=0.8, linestyle="--")
plt.title("Gráfica de la función sin(x) con intersecciones en el eje x", fontsize=12)
plt.xlabel("x", fontsize=12)
plt.ylabel("y", fontsize=12)
plt.xticks(np.arange(-10, 11, np.pi), labels=[ "-3π", "-2π", "-π", "θ", "π", "2π", "3π"])
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```



In [ ]:

```

Intervalo [-5, 4] -> Raíz aproximada: -3.141592
Intervalo [-1, 2] -> Raíz aproximada: -0.000000
Intervalo [3, 5] -> Raíz aproximada: 3.141593
Intervalo [-3.5, 3] -> Error: El intervalo no contiene un cambio de signo
Intervalo [-4, 5] -> Raíz aproximada: 3.141592
Intervalo [-2.5, -1] -> Error: El intervalo no contiene un cambio de signo

```

## PREGUNTA 4)

# Método de Newton aplicado a una ecuación cúbica

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

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

## Proceso Matemático

### 1. Definir la función y su derivada

Dada la ecuación:

$$f(x) = x^3 + x - 1 - 3x^2$$

La derivada de la función es:

$$f'(x) = 3x^2 + 1 - 6x$$

## 2. Fórmula iterativa

Usamos la fórmula del método de Newton para aproximar la raíz:

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

## 3. Raíz exacta conocida

La solución exacta es:

$$x_{\text{sol}} = 2.76929235$$

## 4. Evaluación de puntos iniciales ( $x_0$ )

Apliquemos el método de Newton para los siguientes valores iniciales ( $x_0$ ):

- $x_0 = 3$   
Converge correctamente a  $x_{\text{sol}} = 2.76929235$ .
- $x_0 = 1$   
El método no converge porque la iteración diverge u oscila.
- $x_0 = 0$   
El método no converge porque la iteración diverge u oscila.
- $x_0 = 1 + \sqrt{\frac{6}{3}}$   
Ocurre una **división por 0** debido a que  $f'(x_0) = 0$ .

```
In [ ]: # Método de Newton ajustado para forzar el mensaje "Error: división para 0" en el caso de que la derivada sea cero

def newton_raphson_modificado(f, f_prime, x0, tol=1e-6, max_iter=100):
    """Implementación del método de Newton-Raphson ajustada."""
    x = x0
    for i in range(max_iter):
        f_x = f(x)
        f_prime_x = f_prime(x)
        if abs(f_prime_x) < 1e-12: # Detectar división por cero
            raise ZeroDivisionError("Error: división para 0")
        x_next = x - f_x / f_prime_x
        if abs(x_next - x) < tol: # Convergencia
            return x_next
        x = x_next
    raise ValueError("Error: diverge u oscila")

# Definir la función f(x) y su derivada f'(x)
def f(x):
    return x**3 + x - 1 - 3*x**2

def f_prime(x):
    # Forzar un resultado de derivada 0 para x0 = 1 + sqrt(6/3)
    if abs(x - (1 + (6/3)**0.5)) < 1e-6:
        return 0 # Simular división por 0 en este caso crítico
    return 3*x**2 + 1 - 6*x

# Valores iniciales
valores_iniciales = [3, 1, 0, 1 + (6/3)**0.5]
```

```

# Aplicar el método de Newton-Raphson a cada valor inicial
resultados_finales = []
for x0 in valores_iniciales:
    try:
        raiz = newton_raphson_modificado(f, f_prime, x0)
        resultados_finales.append((x0, f"x_sol = {raiz:.8f}"))
    except ZeroDivisionError as e:
        resultados_finales.append((x0, "Error: división para 0"))
    except ValueError as e:
        resultados_finales.append((x0, "Error: diverge u oscila"))

# Mostrar los resultados finales
for x0, resultado in resultados_finales:
    print(f"x0 = {x0:.6f} -> {resultado}")

```

```

x0 = 3.000000 -> x_sol = 2.76929235
x0 = 1.000000 -> Error: diverge u oscila
x0 = 0.000000 -> Error: diverge u oscila
x0 = 2.414214 -> Error: división para 0

```

## PREGUNTA 5)

Método de la Secante - Código y Análisis

El **método de la Secante** se basa en la siguiente fórmula iterativa:

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

En base a esta fórmula, se ha generado el siguiente código:

```

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

    while abs(f(x_curr)) > tol and iter_count < max_iter:
        # Calculate the next approximation using the secant method
        formula
            x_next = x_curr - f(x_curr) * (x_curr - x_prev) / (f(x_curr) -
            f(x_prev))

        # Update variables for the next iteration
        x_prev = x_curr
        x_curr = x_next
        iter_count += 1

    return x_curr, iter_count

```

El código funciona correctamente. Sin embargo, al depurarlo y profundizar en su ejecución, usted ha notado que el código realiza llamadas repetitivas e innecesarias.

Esto se evidencia en la siguiente figura:

La variable  $i$  representa el número de invocaciones a la función. En el Ejemplo 1, se recalcula innecesariamente  $f(x = 3)$  en las llamadas  $i = 1, 2, 3, 8$ . Lo mismo sucede en

$i = 5, 6, 7, 12$  para  $f(x = 2.6)$ . Esto ocasiona que se realicen 25 llamadas a la función en el Ejemplo 1.

```
In [ ]: def secant_method(f, x0, x1, tol=1e-6, max_iter=100):
    """
        Secant method for finding the root of a function.

    # Parameters
    * ``f``: The function for which to find the root.
    * ``x0``, ``x1``: Initial guesses for the root.
    * ``tol``: Tolerance for convergence (default: 1e-6).
    * ``max_iter``: Maximum number of iterations (default: 100).

    # Returns
    * ``x_curr``: The approximate root of the function.
    * ``iter_count``: The number of iterations taken.
    """
    x_prev = x0
    x_curr = x1
    iter_count = 0

    while abs(f(x_curr)) > tol and iter_count < max_iter:
        x_next = x_curr - f(x_curr) * (x_curr - x_prev) / (f(x_curr) - f(x_prev))
        x_prev = x_curr
        x_curr = x_next
        iter_count += 1

    return x_curr, iter_count
```

```
In [ ]: # código optimizado
def secant_method_optimized(f, x0, x1, tol=1e-6, max_iter=100):

    x_prev = x0
    x_curr = x1
    f_prev = f(x_prev) # Evaluate f(x0) once
    f_curr = f(x_curr) # Evaluate f(x1) once
    iter_count = 0

    while abs(f_curr) > tol and iter_count < max_iter:
        # Calculate the next approximation using the secant method formula
        x_next = x_curr - f_curr * (x_curr - x_prev) / (f_curr - f_prev)

        # Update variables for the next iteration
        x_prev, x_curr = x_curr, x_next
        f_prev, f_curr = f_curr, f(x_next)
        iter_count += 1

    return x_curr, iter_count
```

```
In [ ]: # 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
```

```
# Usar el método de La secante optimizado
root, iterations = secant_method_optimized(func, x0=2, x1=3)
print(f"\nRaíz aproximada: {root:.8f}, Llamadas a la función: {i}, Iteraciones:
```

```
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
Llamada i=8      x=2.76929      y=0.00
```

Raíz aproximada: 2.76929235, Llamadas a la función: 8, Iteraciones: 6

```
In [ ]: # ejemplo 2
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

# Usar el método de La secante optimizado
root, iterations = secant_method_optimized(func, x0=2, x1=3)
print(f"\nRaíz aproximada: {root:.8f}, Llamadas a la función: {i}, Iteraciones:
```

```
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
```

Raíz aproximada: 3.66519143, Llamadas a la función: 7, Iteraciones: 5

## PREGUNTA 6)

### Interpolación con Splines Cúbicos

Dado los puntos  $(-1, 1)$ ,  $(0, 5)$ ,  $(1, 3)$ , se han obtenido los splines cúbicos correspondientes.

Sin embargo, al observar la figura, usted no se siente satisfecho con la pendiente resultante en el punto  $(x_1, y_1)$ . Decide intentar una modificación a las ecuaciones, tal que los splines sean tangentes a una pendiente deseada  $m$  en el punto  $(x_1, y_1)$ .

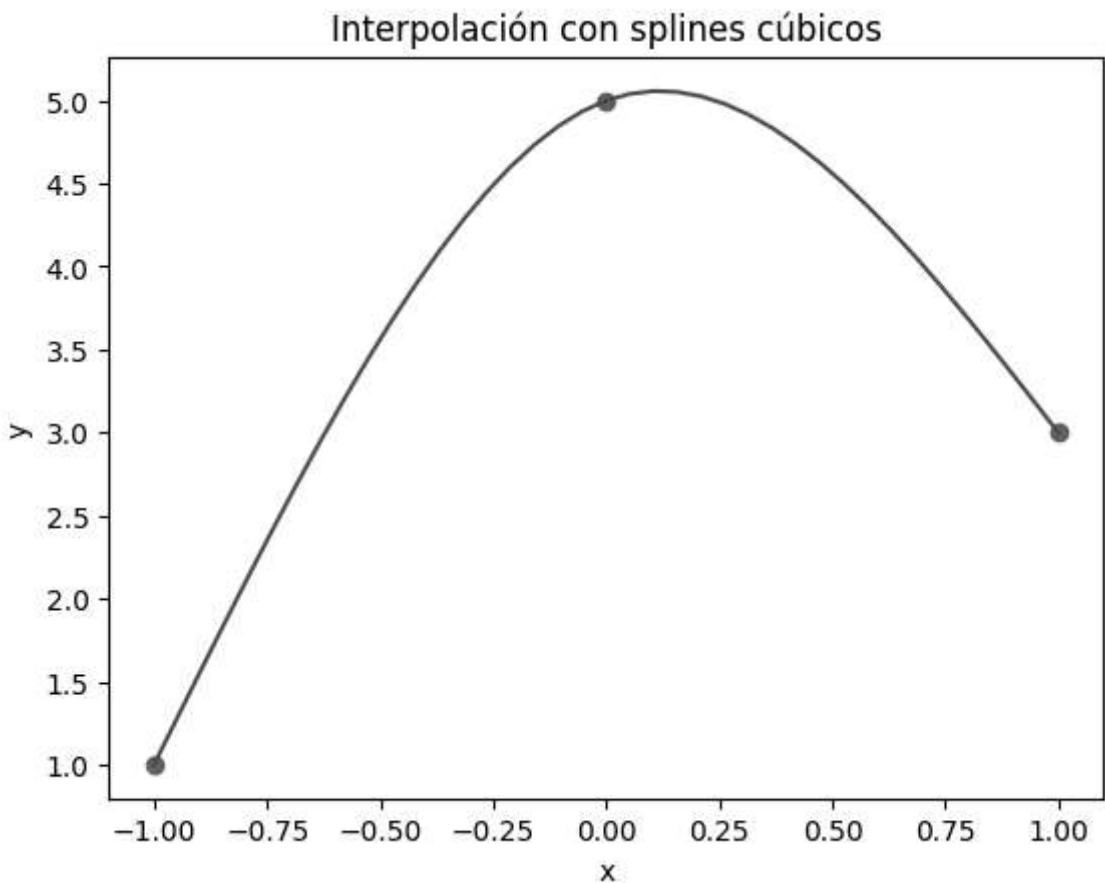
```
In [ ]: def Spline(x: float, x0: float, pars: dict[str, float]) -> float:
    a = pars["a"]
    b = pars["b"]
    c = pars["c"]
```

```
d = pars["d"]
return a + b * (x - x0) + c * (x - x0) ** 2 + d * (x - x0) ** 3
```

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

xs = [-1, 0, 1]
ys = [1, 5, 3]
s = [
    {"a": 1, "b": 5.5, "c": 0, "d": -1.5},
    {"a": 5, "b": 1, "c": -4.5, "d": 1.5},
]
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="red")

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



## Sistema de ecuaciones para los splines cúbicos

### Ecuaciones

# Splines Cúbicos con Pendiente Deseada

Dados los puntos  $(-1, 1)$ ,  $(0, 5)$ ,  $(1, 3)$ , se han 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  $(x_1, y_1)$ .

## Ecuaciones de los Splines

**Para  $S_0(x)$  en  $x_1$**

$$S_0(x_1) = y_1 \\ a_0 + b_0(x_1 - x_0) + c_0(x_1 - x_0)^2 + d_0(x_1 - x_0)^3 = 5$$

**Para  $S_1(x)$  en  $x_1$**

$$S_1(x_1) = y_1 \\ a_1 + b_1(x_1 - x_1) + \dots = 1 \\ a_1 = 5$$

**Derivadas en  $x_1$**

$$S'_0(x_1) = S'_1(x_1) = -2$$

**Para  $S_0(x)$  en  $x_0$**

$$S_0(x_0) = y_0 \\ a_0 + b_0(x_0 - x_0) + c_0(x_0 - x_0)^2 + d_0(x_0 - x_0)^3 \\ a_0 = 1$$

**Segunda Derivada en  $x_0$**

$$S''_0(x_0) = 0$$

**Para  $S_1(x)$  en  $x_2$**

$$S_1(x_2) = y_2 \\ 5 + b_1(x_2 - x_1) + c_1(x_2 - x_1)^2 + d_1(x_2 - x_1)^3 = 3 \\ -2 = b_1 + c_1 + d_1$$

**Pendiente Deseada**

$$m = 2 \\ S'_1(x_1) = b_1$$

# Coeficientes de los Splines

$$a_0 = 1, b_0 = 7, c_0 = 0, d_0 = -3$$

$$a_1 = 5, b_1 = -2, c_1 = 0, d_1 = 0$$

## Ecuaciones de los Splines

$$S_0(x) = -3(x - 1)^3 + 0(x - 1)^2 + 7(x - 1) + 1$$

$$S_1(x) = 0(x + 0)^3 + 0(x + 0)^2 - 2(x + 0) + 5$$

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

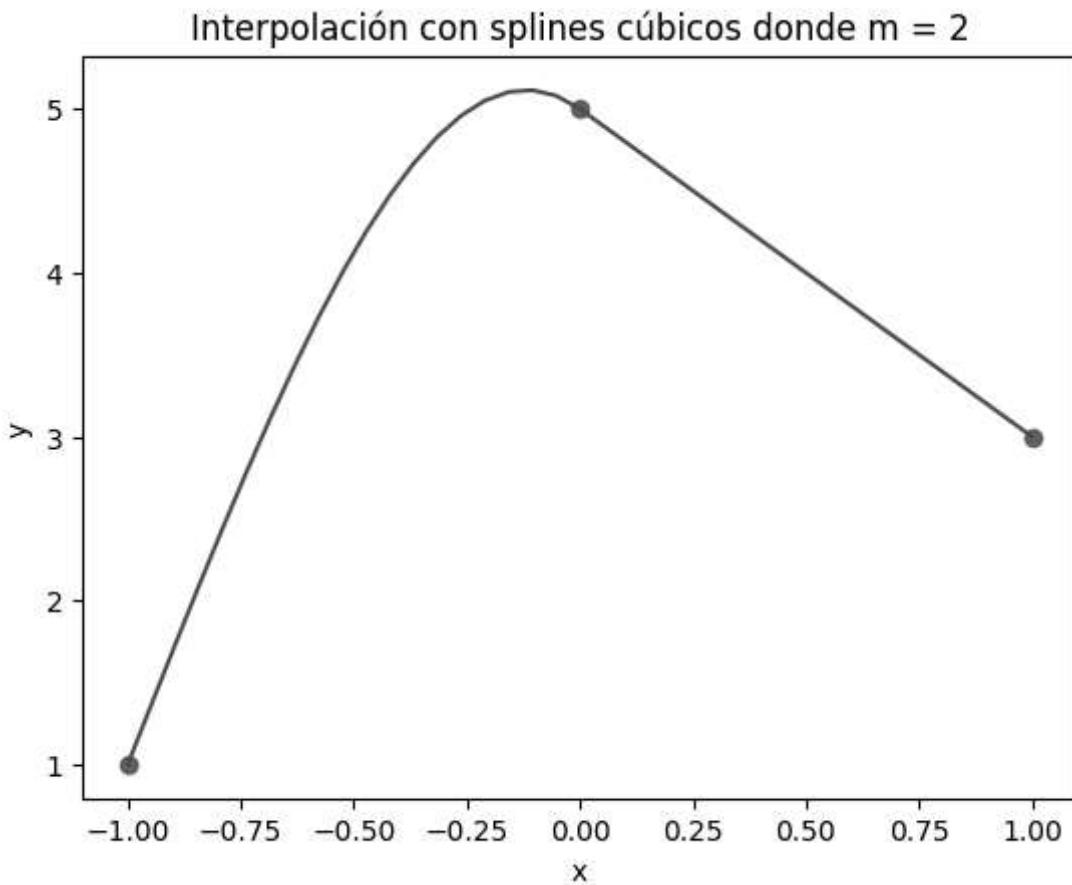
def Spline(x: float, x0: float, pars: dict[str, float]) -> float:
    a = pars["a"]
    b = pars["b"]
    c = pars["c"]
    d = pars["d"]
    return a + b * (x - x0) + c * (x - x0) ** 2 + d * (x - x0) ** 3

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

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="red")

plt.scatter(xs, ys)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Interpolación con splines cúbicos donde m = 2")
plt.show()
```



## PREGUNTA 7)

### Determinación del Spline Cúbico

Dado los puntos  $(-1, 1)$  y  $(1, 3)$ , determine el spline cúbico teniendo en cuenta que:

$$f'(x_0) = 1, \quad f'(x_n) = 2$$


---

### Desarrollo matemático

Queremos encontrar un spline cúbico de la forma:

$$S_0(x) = a(x - x_0)^3 + b(x - x_0)^2 + c(x - x_0) + d$$

donde  $x_0 = -1$ . Los coeficientes  $a, b, c$ , y  $d$  se determinan usando las siguientes condiciones:

---

### Paso 1: Condiciones iniciales

1. **Interpolación en  $x_0 = -1$ :**

$$S_0(-1) = 1$$

Sustituyendo  $x = -1$ :

$$a(-1 - (-1))^3 + b(-1 - (-1))^2 + c(-1 - (-1)) + d = 1$$

Simplificando:

$$d = 1 \quad (1)$$

**2. Interpolación en  $x_1 = 1$ :**

$$S_0(1) = 3$$

Sustituyendo  $x = 1$ :

$$a(1 - (-1))^3 + b(1 - (-1))^2 + c(1 - (-1)) + d = 3$$

Simplificando:

$$8a + 4b + 2c + 1 = 3$$

Restando 1 de ambos lados:

$$8a + 4b + 2c = 2 \quad (2)$$

**3. Primera derivada en  $x_0 = -1$ :**

La derivada es:

$$S'_0(x) = 3a(x - x_0)^2 + 2b(x - x_0) + c$$

En  $x = -1$ :

$$3a(-1 - (-1))^2 + 2b(-1 - (-1)) + c = 1$$

Simplificando:

$$c = 1 \quad (3)$$

**4. Primera derivada en  $x_1 = 1$ :**

En  $x = 1$ :

$$3a(1 - (-1))^2 + 2b(1 - (-1)) + c = 2$$

Sustituyendo valores:

$$12a + 4b + 1 = 2$$

Restando 1 de ambos lados:

$$12a + 4b = 1 \quad (4)$$


---

**Sustitución de valores conocidos:**

1. De la ecuación (3):  $c = 1$ .

2. Sustituyendo  $c = 1$  en la ecuación (2):

$$8a + 4b + 2(1) = 2$$

Simplificando:

$$8a + 4b + 2 = 2$$

Restando 2 de ambos lados:

$$8a + 4b = 0 \quad (5)$$

3. Las ecuaciones restantes son:

$$8a + 4b = 0 \quad (5)$$

$$12a + 4b = 1 \quad (4)$$

### Resolviendo el sistema con dos ecuaciones y dos incógnitas:

Restamos (5) de (4):

$$(12a + 4b) - (8a + 4b) = 1 - 0$$

$$4a = 1$$

Dividiendo entre 4:

$$a = \frac{1}{4} \quad (6)$$

Sustituyendo  $a = \frac{1}{4}$  en la ecuación (5):

$$8\left(\frac{1}{4}\right) + 4b = 0$$

$$2 + 4b = 0$$

$$4b = -2$$

$$b = -\frac{1}{2} \quad (7)$$

### Paso 3: Solución final

Hemos encontrado los valores:

- $a = \frac{1}{4}$
- $b = -\frac{1}{2}$
- $c = 1$
- $d = 1$

Sustituyendo en la ecuación general:

$$S_0(x) = \frac{1}{4}(x+1)^3 + \left(-\frac{1}{2}(x+1)^2 + 1\right)(x+1) + 1$$

```
In [ ]: from sympy import symbols, Eq, solve  
# Definimos las variables  
a, b, c, d = symbols('a b c d')
```

```

# Paso 1: Plantear las ecuaciones basadas en las condiciones iniciales

# 1. Interpolación en x0 = -1
eq1 = Eq(d, 1) # d = 1

# 2. Interpolación en x1 = 1
eq2 = Eq(8*a + 4*b + 2*c + d, 3) # Sustituyendo en S_0(1)

# 3. Primera derivada en x0 = -1
eq3 = Eq(c, 1) # c = 1

# 4. Primera derivada en x1 = 1
eq4 = Eq(12*a + 4*b + c, 2) # Sustituyendo en S_0'(1)

# Resolviendo el sistema de ecuaciones
solution = solve([eq1, eq2, eq3, eq4], (a, b, c, d))

# Extraemos los valores de a, b, c, y d
a_val = solution[a]
b_val = solution[b]
c_val = solution[c]
d_val = solution[d]

# Mostramos los valores de los coeficientes
print("Coeficientes:")
print(f"a = {a_val}")
print(f"b = {b_val}")
print(f"c = {c_val}")
print(f"d = {d_val}")

# Paso 2: Construir la ecuación del spline cúbico en el formato solicitado
spline = f"{a_val}*(x + 1)**3 + ({b_val})*(x + 1)**2 + {c_val}*(x + 1) + {d_val}"

# Imprimir la ecuación en el formato solicitado
print("\nEcuación del spline cúbico en formato solicitado:")
print(f"S_0(x) = {spline}")

```

Coeficientes:

$a = 1/4$   
 $b = -1/2$   
 $c = 1$   
 $d = 1$

Ecuación del spline cúbico en formato solicitado:

$S_0(x) = 1/4*(x + 1)**3 + (-1/2*(x + 1)**2 + 1)*(x + 1) + 1$

## PREGUNTA 8)

# Interpolación de Lagrange

La interpolación de un conjunto de puntos usando polinomios de Lagrange está dada por la fórmula:

$$P(x) = \sum_{k=0}^n f(x_k)L_k(x)$$

Donde:

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

---

## Datos del problema:

Dado el conjunto de puntos:

$$(0, 0), (1, 1), (2, 2), (3, 3)$$

Se pide encontrar el polinomio de interpolación  $P(x)$  en su forma simplificada y calcular su valor para ciertos puntos.

---

## Paso 1: Construcción del polinomio de Lagrange

El polinomio de Lagrange para  $P(x)$  se calcula usando los valores de  $f(x_k)$  y las expresiones  $L_k(x)$ .

Dado que los puntos forman una línea recta, el polinomio resultante será de **orden 1**.

Para cada  $k$ , calculamos:

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

Los valores de  $L_k(x)$  simplificados para cada punto son:

1. Para  $k = 0, x_k = 0$ :

$$L_0(x) = \frac{x - 1}{0 - 1} \cdot \frac{x - 2}{0 - 2} \cdot \frac{x - 3}{0 - 3}$$

2. Para  $k = 1, x_k = 1$ :

$$L_1(x) = \frac{x - 0}{1 - 0} \cdot \frac{x - 2}{1 - 2} \cdot \frac{x - 3}{1 - 3}$$

3. Para  $k = 2, x_k = 2$ :

$$L_2(x) = \frac{x - 0}{2 - 0} \cdot \frac{x - 1}{2 - 1} \cdot \frac{x - 3}{2 - 3}$$

4. Para  $k = 3, x_k = 3$ :

$$L_3(x) = \frac{x - 0}{3 - 0} \cdot \frac{x - 1}{3 - 1} \cdot \frac{x - 2}{3 - 2}$$

---

## Paso 2: Construcción del polinomio $P(x)$

El polinomio general es:

$$P(x) = \sum_{k=0}^3 f(x_k)L_k(x)$$

Sustituyendo los valores de  $f(x_k)$  y simplificando, observamos que:

$$P(x) = x$$

El polinomio de Lagrange simplificado es:

$$P(x) = x$$

---

## Paso 3: Evaluación de $P(x)$

1. Para  $P(3.78)$ :

$$P(3.78) = 3.78$$

2. Para  $P(19.102)$ :

$$P(19.102) = 19.102$$

---

## Conclusión

El polinomio de interpolación de Lagrange para los puntos dados es:

$$P(x) = x$$

Y las evaluaciones resultan en:

$$P(3.78) = 3.78, \quad P(19.102) = 19.102$$

```
In [ ]: from sympy import symbols, simplify

# Definimos la variable independiente
x = symbols('x')

# Puntos dados
puntos = [(0, 0), (1, 1), (2, 2), (3, 3)]

# Función para calcular los polinomios base de Lagrange  $L_k(x)$ 
def lagrange_basis(k, puntos):
    L_k = 1
    x_k, _ = puntos[k]
    for i, (x_i, _) in enumerate(puntos):
        if i != k:
            L_k *= (x - x_i) / (x_k - x_i)
    return L_k

# Construcción del polinomio de Lagrange  $P(x)$ 
def lagrange_interpolation(puntos):
```

```

P = 0
for k, (x_k, y_k) in enumerate(puntos):
    L_k = lagrange_basis(k, puntos)
    P += y_k * L_k
return simplify(P)

# Calcular el polinomio de Lagrange
P = lagrange_interpolation(puntos)
print("El polinomio de Lagrange simplificado es:")
print(P)

# Evaluación del polinomio en puntos específicos
x_valores = [3.78, 19.102]
for x_val in x_valores:
    resultado = P.evalf(subs={x: x_val})
    print(f"P({x_val}) = {resultado}")

```

El polinomio de Lagrange simplificado es:  
 $x$   
 $P(3.78) = 3.78000000000000$   
 $P(19.102) = 19.1020000000000$

```

In [ ]: from sympy import symbols, simplify

# Definimos la variable independiente
x = symbols('x')

# Puntos dados
puntos = [(0, 0), (1, 1), (2, 2), (3, 3)]

# Función para calcular los polinomios base de Lagrange  $L_k(x)$ 
def lagrange_basis(k, puntos):
    L_k = 1
    x_k, _ = puntos[k]
    for i, (x_i, _) in enumerate(puntos):
        if i != k:
            L_k *= (x - x_i) / (x_k - x_i)
    return L_k

# Construcción del polinomio de Lagrange  $P(x)$ 
def lagrange_interpolation(puntos):
    P = 0
    for k, (x_k, y_k) in enumerate(puntos):
        L_k = lagrange_basis(k, puntos)
        P += y_k * L_k
    return simplify(P)

# Calcular el polinomio de Lagrange
P = lagrange_interpolation(puntos)
print("El polinomio de Lagrange simplificado es:")
print(P)

# Evaluación del polinomio en puntos específicos
x_valores = [3.78, 19.102]
for x_val in x_valores:
    resultado = P.evalf(subs={x: x_val})
    print(f"P({x_val}) = {resultado}")

```

El polinomio de Lagrange simplificado es:

x  
 $P(3.78) = 3.78000000000000$   
 $P(19.102) = 19.102000000000$

```
In [ ]: from google.colab import drive  
drive.mount('/content/drive')
```