Actividad extracurricular 08 - Corrección Examen

Nombre: Alexis Bautista
Fecha de entrega: 10 de diciembre de 2024
Paralelo: GR1CC
Enlace de GitHub: https://github.com/alexis-bautista/CorrecionExamen01-MN.git

Pregunta 1

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

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

Calcule el error relativo en las siguientes aproximaciones de π mediante el polinomio (en lugar del arcotangente).
Asuma que $\pi = 3.14159$.

$$4(\arctan\frac{1}{2} + \arctan\frac{1}{3})$$

Dado que tenemos una ecuación de quinto orden:

$4(\arctan\frac{1}{2} + \arctan\frac{1}{3}) = \pi$

$4([\frac{1}{2} - \frac{1}{3}(\frac{1}{2})^3 + \frac{1}{5}(\frac{1}{2})^5] + [\frac{1}{3} - \frac{1}{3}(\frac{1}{3})^3 + \frac{1}{5}(\frac{1}{3})^5])$

simplificamos la ecuación

$= 4(\frac{223}{480} + \frac{391}{1215})$

$\approx 3.145576132 = \pi$

calculamos el error relativo

$\epsilon = \left|\frac{\text{Valor aproximado} - \text{Valor verdadero}}{\text{Valor verdadero}}\right|$

$\epsilon = \frac{3.14159 - 3.145576132}{3.145576132}$

$= 1.267218 \times 10^{-3}$

≈ 0.001267

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

$\epsilon = 0.001267$

¿En qué orden de magnitud está este error? Es decir, $\epsilon < 10^n, n =$ -3

$$16 * \arctan \frac{1}{5} - 4 * \arctan \frac{1}{239}$$

Dado que tenemos una ecuación de quinto orden:

$16 * \arctan \frac{1}{5} - 4 * \arctan \frac{1}{239} = \pi$

$= 16[\frac{1}{5} - \frac{1}{3}(\frac{1}{5})^3 + \frac{1}{5}(\frac{1}{5})^5] - 4[\frac{1}{239} - \frac{1}{3}(\frac{1}{239})^3 + \frac{1}{5}(\frac{1}{239})^5]$

$\approx 16(\frac{9253}{46875}) - 4(4.184076 \times 10^{-6})$

$\approx 3.158357333 - 0.016736304$

$\approx 3.141621029 = \pi$

calculamos el error relativo

$\epsilon = \left|\frac{\text{Valor aproximado} - \text{Valor verdadero}}{\text{Valor verdadero}}\right|$

$\epsilon = \frac{3.14159 - 3.141621029}{3.141621029}$

$= 9.87675 \times 10^{-6}$

≈ 0.000009877

$\epsilon \approx 0.000010$

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

Código

$4(\arctan \frac{1}{2} + \arctan \frac{1}{3})$:

In [3]:
```
import math

# Serie de Maclaurin para arctan(x) (3 primeros términos)
def arctan_maclaurin(x):
    return x - (x**3) / 3 + (x**5) / 5

# Calcular arctan(1/2) y arctan(1/3) usando la serie de Maclaurin
arctan_1_2 = arctan_maclaurin(1/2)
arctan_1_3 = arctan_maclaurin(1/3)

# Calcular 4 * (arctan(1/2) + arctan(1/3))
approx_pi = 4 * (arctan_1_2 + arctan_1_3)

# Valor verdadero de pi
true_pi = 3.14159

# Calcular el error relativo
```

```python
error_relative = abs((approx_pi - true_pi) / true_pi)

# Determinar el orden de magnitud del error
order_of_magnitude = math.floor(math.log10(error_relative))

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

```
Error relativo: 0.001269
El error está en el orden de magnitud: 10^-3
```

$16 * \arctan \frac{1}{5} - 4 * \arctan \frac{1}{239}$

In [4]:
```python
# Calcular arctan(1/5) y arctan(1/239) usando la serie de Maclaurin
arctan_1_5 = arctan_maclaurin(1/5)
arctan_1_239 = arctan_maclaurin(1/239)

# Calcular 4 * (arctan(1/5) + arctan(1/239))
approx_pi_2 = 16 * arctan_1_5 - 4 * arctan_1_239

# Calcular el error relativo
error_relative = abs((approx_pi_2 - true_pi) / true_pi)

# Determinar el orden de magnitud del error
order_of_magnitude = math.floor(math.log10(error_relative))

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

```
Error relativo: 0.000010
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: $x = \frac{x_0 * y_1 - x_1 * y_0}{y_1 - y_0}$
y
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):

Usamos el método A como base para calcular el valor real

$x = \frac{1.31 * 4.76 - 1.93 * 3.24}{4.76 - 3.24}$

$= -0.01167894737$

$x = -0.0115789$

Usando aritmética de computador con redondeo a 3 cifras significativas resuelva para ambos métodos.

Usando el métdodo A:

$$x = \frac{1.31*4.76-1.93*3.24}{4.76-3.24}$$

$$\approx \frac{6.24-6.25}{4.76-3.24}$$

$$\approx -\frac{0.01}{1.52} \approx -6.5789 \times 10^{-3}$$

$$\approx -0.00658$$

El error relativo (redondee al final del cálculo a 3 cifras significativas) del método A:

$$\epsilon = \left|\frac{\text{Valor aproximado} - \text{Valor verdadero}}{\text{Valor verdadero}}\right|$$

$$\epsilon = \frac{-0.00658+0.0115789}{0.0115789}$$

$$\epsilon \approx 0.432$$

Usando el método B:

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

$$\approx 1.31 - \frac{(0.62)*3.24}{1.52}$$

$$\approx 1.31 - \frac{2.01}{1.52}$$

$$\approx 1.31 - 1.32 \approx -0.01$$

El error relativo (redondee al final del cálculo a 3 cifras significativas) del método B:

$$\epsilon = \left|\frac{\text{Valor aproximado} - \text{Valor verdadero}}{\text{Valor verdadero}}\right|$$

$$\epsilon = \frac{-0.01+0.0115789}{0.0115789}$$

$$\epsilon \approx 0.136$$

¿Cuál método es mejor?

El método B es mejor porque tiene un error relativo más bajo (13.6%) en comparación con el método A (43.2%).

También podemos notar que el método b es mejor ya que tiene menos multiplicaciones, por lo tanto tendremos un error mas bajo.

Código

In [18]:
```
import numpy as np
```

```python
# Datos iniciales
x0, y0 = 1.31, 3.24
x1, y1 = 1.93, 4.76

# Valor real (Método A con alta precisión)
def metodo_a_preciso(x0, y0, x1, y1):
    return (x0 * y1 - x1 * y0) / (y1 - y0)

valor_real = metodo_a_preciso(x0, y0, x1, y1)

# Funciones para métodos con redondeo a 3 cifras significativas
def round_sig(x, sig=3):
    return round(x, sig - int(np.floor(np.log10(abs(x)))) - 1)

def metodo_a(x0, y0, x1, y1):
    num = round_sig(x0 * y1 - x1 * y0)
    den = round_sig(y1 - y0)
    return round_sig(num / den)

def metodo_b(x0, y0, x1, y1):
    diff_x = round_sig(x1 - x0)
    num = round_sig(diff_x * y0)
    den = round_sig(y1 - y0)
    return round_sig(x0 - num / den)

# Calcular aproximaciones
aprox_a = metodo_a(x0, y0, x1, y1)
aprox_b = metodo_b(x0, y0, x1, y1)

# Calcular errores relativos
def error_relativo(aprox, real):
    return abs((aprox - real) / real)

error_a = round_sig(error_relativo(aprox_a, valor_real))
error_b = round_sig(error_relativo(aprox_b, valor_real))

# Imprimir resultados
print("Resultados del cálculo de intersección:")
print(f"Valor real (preciso, Método A): x = {valor_real:.6f}")
print(f"Método A: x = {aprox_a}, Error relativo: {error_a:.3f}")
print(f"Método B: x = {aprox_b}, Error relativo: {error_b:.3f}")

# Conclusión sobre el mejor método
if error_a < error_b:
    print("El método A es mejor debido a un error relativo más bajo.")
else:
    print("El método B es mejor debido a un error relativo más bajo.")
```

```
Resultados del cálculo de intersección:
Valor real (preciso, Método A): x = -0.011579
Método A: x = -0.0116, Error relativo: 0.002
Método B: x = -0.0124, Error relativo: 0.071
El método A es mejor debido a un error relativo más bajo.
```

Pregunta 3

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 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.

```python
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)
```

[3] ✓ 0.0s

```
Llamada i=1     x=3.00000     y=2.00
Llamada i=2     x=3.00000     y=2.00
Llamada i=3     x=3.00000     y=2.00
Llamada i=4     x=2.00000     y=-3.00
Llamada i=5     x=2.60000     y=-1.10
Llamada i=6     x=2.60000     y=-1.10
Llamada i=7     x=2.60000     y=-1.10
Llamada i=8     x=3.00000     y=2.00
Llamada i=9     x=2.74227     y=-0.20
Llamada i=10    x=2.74227     y=-0.20
Llamada i=11    x=2.74227     y=-0.20
Llamada i=12    x=2.60000     y=-1.10
Llamada i=13    x=2.77296     y=0.03
Llamada i=14    x=2.77296     y=0.03
Llamada i=15    x=2.77296     y=0.03
Llamada i=16    x=2.74227     y=-0.20
Llamada i=17    x=2.76922     y=-0.00
Llamada i=18    x=2.76922     y=-0.00
Llamada i=19    x=2.76922     y=-0.00
Llamada i=20    x=2.77296     y=0.03
Llamada i=21    x=2.76929     y=-0.00
Llamada i=22    x=2.76929     y=-0.00
Llamada i=23    x=2.76929     y=-0.00
Llamada i=24    x=2.76922     y=-0.00
Llamada i=25    x=2.76929     y=0.00
```

```
(2.7692923542484045, 6)
```

Modifique el código provisto para optimizar el número de llamadas a la función.

https://github.com/ztjona/MN-examen-01-2024-B/blob/main/secante_optimizar.ipynb

---

Función dada

In [5]:

```python
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
```

Ejemplo 1:

In [6]:
```
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=3.00000	 y=2.00
Llamada i=2	 x=3.00000	 y=2.00
Llamada i=3	 x=3.00000	 y=2.00
Llamada i=4	 x=2.00000	 y=-3.00
Llamada i=5	 x=2.60000	 y=-1.10
Llamada i=6	 x=2.60000	 y=-1.10
Llamada i=7	 x=2.60000	 y=-1.10
Llamada i=8	 x=3.00000	 y=2.00
Llamada i=9	 x=2.74227	 y=-0.20
Llamada i=10	 x=2.74227	 y=-0.20
Llamada i=11	 x=2.74227	 y=-0.20
Llamada i=12	 x=2.60000	 y=-1.10
Llamada i=13	 x=2.77296	 y=0.03
Llamada i=14	 x=2.77296	 y=0.03
Llamada i=15	 x=2.77296	 y=0.03
Llamada i=16	 x=2.74227	 y=-0.20
Llamada i=17	 x=2.76922	 y=-0.00
Llamada i=18	 x=2.76922	 y=-0.00
Llamada i=19	 x=2.76922	 y=-0.00
Llamada i=20	 x=2.77296	 y=0.03
Llamada i=21	 x=2.76929	 y=-0.00
Llamada i=22	 x=2.76929	 y=-0.00
Llamada i=23	 x=2.76929	 y=-0.00
Llamada i=24	 x=2.76922	 y=-0.00
Llamada i=25	 x=2.76929	 y=0.00
```

Out[6]:
```
(2.7692923542484045, 6)
```

Ejemplo 2:

In [7]:
```python
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=3.00000      y=2.00
Llamada i=2      x=3.00000      y=2.00
Llamada i=3      x=3.00000      y=2.00
Llamada i=4      x=2.00000      y=-3.00
Llamada i=5      x=2.60000      y=-1.10
Llamada i=6      x=2.60000      y=-1.10
Llamada i=7      x=2.60000      y=-1.10
Llamada i=8      x=3.00000      y=2.00
Llamada i=9      x=2.74227      y=-0.20
Llamada i=10     x=2.74227      y=-0.20
Llamada i=11     x=2.74227      y=-0.20
Llamada i=12     x=2.60000      y=-1.10
Llamada i=13     x=2.77296      y=0.03
Llamada i=14     x=2.77296      y=0.03
Llamada i=15     x=2.77296      y=0.03
Llamada i=16     x=2.74227      y=-0.20
Llamada i=17     x=2.76922      y=-0.00
Llamada i=18     x=2.76922      y=-0.00
Llamada i=19     x=2.76922      y=-0.00
Llamada i=20     x=2.77296      y=0.03
Llamada i=21     x=2.76929      y=-0.00
Llamada i=22     x=2.76929      y=-0.00
Llamada i=23     x=2.76929      y=-0.00
Llamada i=24     x=2.76922      y=-0.00
Llamada i=25     x=2.76929      y=0.00
```

Out[7]: (2.7692923542484045, 6)

Función optimizada

In [8]:
```
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

    f_x_prev = f(x_prev)
    f_x_curr = f(x_curr)

    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

        f_x_prev = f_x_curr
        f_x_curr = f(x_curr)

        iter_count += 1
```

```
    return x_curr, iter_count
```

Ejemplo 1:

In [9]:
```
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
Llamada i=8      x=2.76929       y=0.00
```

Out[9]:
```
(2.7692923542484045, 6)
```

Ejemplo 2:

In [10]:
```
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
```

Out[10]:
```
(3.66519143172732, 5)
```

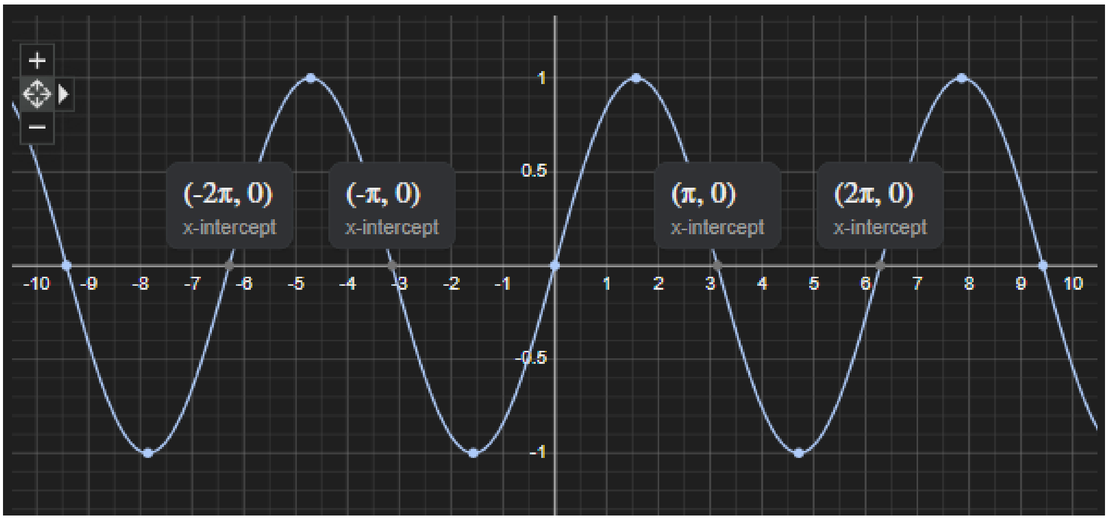
Luego de optimizar el código y utilizando $(x_0 = 2, x_1 = 3)$, conteste:
¿Cuál es el número mínimo de llamadas a la función para llegar a la raíz en el Ejemplo 1?
$i = 8$

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 2?
$i = 7$

La optimización realizada en el código consiste en evitar llamadas redundantes a la función $f(x)$. En el código original, $f(x_{curr})$ y $f(x_{prev})$ se calculaban repetidamente dentro del bucle. En la versión optimizada, estos valores se almacenan en las variables `f_x_curr` y `f_x_prev`, actualizándolos únicamente cuando cambian x_{curr} y x_{prev}, respectivamente. Esto reduce el número de evaluaciones de $f(x)$, mejorando la eficiencia del algoritmo, especialmente si $f(x)$ es costosa de calcular.

Pregunta 4

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



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

1. $a = -1, b = 2$

$\sin -1 * \sin 2 < 0$

La raíz en este intervalo es $x = 0$, ya que esta entre -1 y 2.

2. $a = 3, b = 5$

$\sin 3 * \sin 5 < 0$

La raíz en este intervalo es $x = \pi$, ya que esta entre 3 y 5.

3. $a = -3.5, b = 3$

$\sin -1 * \sin 2 > 0$

En este intervalo no hay cambio de signo, por lo que el método de Bisección no converge a ninguna raíz.

4. $a = -4, b = 5$

$\sin -4 * \sin 5 < 0$

Este intervalo incluye dos raíces ($-\pi$ y π). Por el método de Bisección, converge a $x = \pi$

5. $a = -5, b = 4$

$\sin -5 * \sin 4 < 0$

Este intervalo incluye varias raíces ($-\pi, 0$ y π). Por el método de Bisección, converge a $x = -\pi$

6. $a = -2.5, b = -1$

$\sin -2.5 * \sin -1 > 0$

En este intervalo no hay cambio de signo, por lo que el método de Bisección no converge a ninguna raíz.

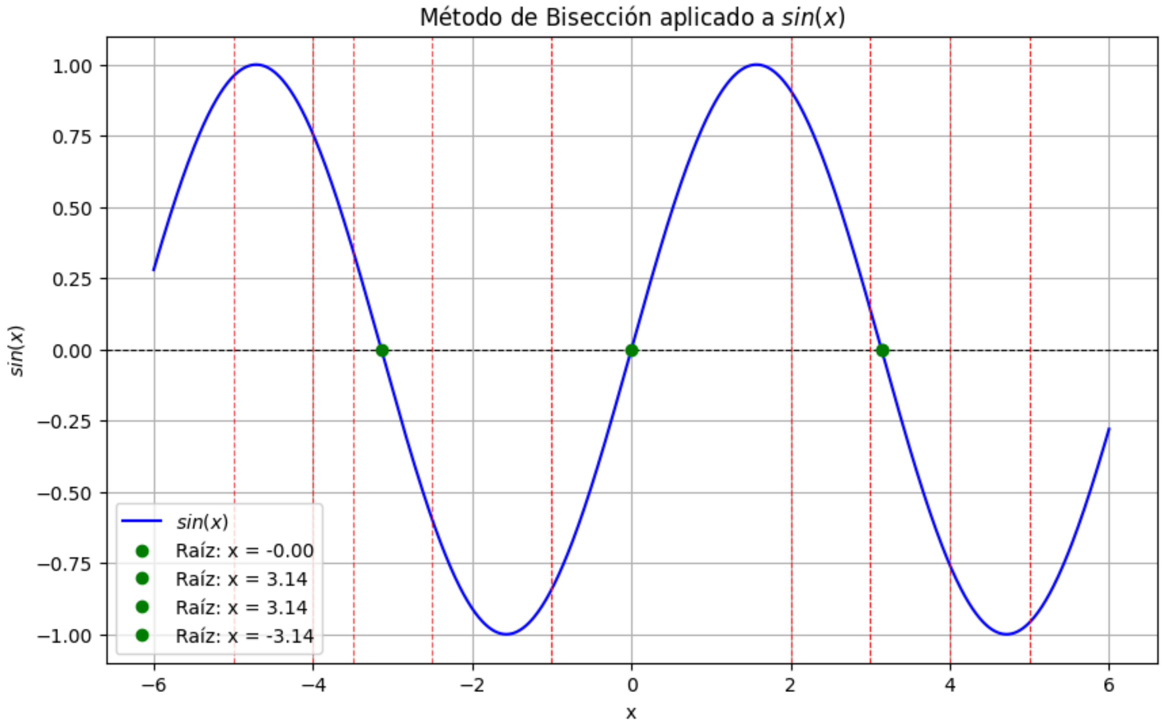
Código

In [17]:

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

# Definimos la función
f = np.sin

def bisection_method(func, a, b, tol=1e-6, max_iter=100):
    """
    Método de la Bisección para encontrar raíces de una función continua.

    Parámetros:
        func: Función para la cual encontrar la raíz.
        a, b: Extremos del intervalo.
        tol: Tolerancia para la convergencia.
        max_iter: Número máximo de iteraciones.

    Retorna:
        (raíz, iteraciones) si hay una raíz, None si no converge.
    """
    if func(a) * func(b) >= 0:
        return None  # No hay cambio de signo en el intervalo

    iter_count = 0
    while (b - a) / 2 > tol and iter_count < max_iter:
        c = (a + b) / 2
        if func(c) == 0:  # Encontramos la raíz exacta
            return c, iter_count
        elif func(a) * func(c) < 0:
            b = c
        else:
            a = c
        iter_count += 1

    return (a + b) / 2, iter_count

# Intervalos para análisis
```

```python
intervalos = [(-1, 2), (3, 5), (-3.5, 3), (-4, 5), (-5, 4), (-2.5, -1)]

# Encontrar raíces en los intervalos dados
resultados = []
for a, b in intervalos:
    resultado = bisection_method(f, a, b)
    resultados.append((a, b, resultado))

# Imprimir resultados
print("Resultados del método de Bisección:")
for a, b, resultado in resultados:
    if resultado is None:
        print(f"Intervalo ({a}, {b}): No converge a ninguna raíz.")
    else:
        raiz, iteraciones = resultado
        print(f"Intervalo ({a}, {b}): Raíz encontrada en x = {raiz:.6f} tras {it

# Graficar la función y los intervalos analizados
x = np.linspace(-6, 6, 1000)
y = f(x)

plt.figure(figsize=(10, 6))
plt.plot(x, y, label="$sin(x)$", color="blue")
plt.axhline(0, color="black", linewidth=0.8, linestyle="--")

# Añadir intervalos y raíces
for a, b, resultado in resultados:
    plt.axvline(a, color="red", linestyle="--", linewidth=0.8, alpha=0.7)
    plt.axvline(b, color="red", linestyle="--", linewidth=0.8, alpha=0.7)
    if resultado is not None:
        raiz, _ = resultado
        plt.plot(raiz, 0, 'o', color='green', label=f"Raíz: x = {raiz:.2f}" if '

# Detalles del gráfico
plt.title("Método de Bisección aplicado a $sin(x)$")
plt.xlabel("x")
plt.ylabel("$sin(x)$")
plt.legend()
plt.grid(True)
plt.show()
```

```
Resultados del método de Bisección:
Intervalo (-1, 2): Raíz encontrada en x = -0.000000 tras 21 iteraciones.
Intervalo (3, 5): Raíz encontrada en x = 3.141593 tras 20 iteraciones.
Intervalo (-3.5, 3): No converge a ninguna raíz.
Intervalo (-4, 5): Raíz encontrada en x = 3.141592 tras 23 iteraciones.
Intervalo (-5, 4): Raíz encontrada en x = -3.141592 tras 23 iteraciones.
Intervalo (-2.5, -1): No converge a ninguna raíz.
```



Pregunta 5

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:

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

Reescribimos la ecuación

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

Derivamos la ecuación

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

$x_{sol} = 2.769292$

Qué sucede cuando:

$x_0 = 3$

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

$f'(3) = 3(3)^2 - 6(3) + 1 = 10$

$x_0 = 3 - \frac{2}{10} = 2.8$

$x_{sol} = 2.8$

cuando:

$x_0 = 1$

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

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

$x_0 = 1 - \frac{-2}{-2} = 0$

Error [diverge u oscila]

cuando:

$x_0 = 0$

$f(0) = 0^3 + 0 - 1 - 3(0)^2 = -1$

$f'(0) = 3(0)^2 - 6(0) + 1 = 1$

$x_0 = 0 - \frac{-1}{1} = 1$

Error [diverge u oscila]

cuando: $x_0 = 1 + \frac{\sqrt{6}}{3}$

$f(1 + \frac{\sqrt{6}}{3}) = (1 + \frac{\sqrt{6}}{3})^3 + (1 + \frac{\sqrt{6}}{3}) - 1 - 3(1 + \frac{\sqrt{6}}{3})^2 \approx -3.088662108$

$f'(1 + \frac{\sqrt{6}}{3}) = 3(1 + \frac{\sqrt{6}}{3})^2 - 6(1 + \frac{\sqrt{6}}{3}) + 1 = 0$

$x_0 = (1 + \frac{\sqrt{6}}{3}) - \frac{-3.088662108}{0} = Error$

Error [division para 0]

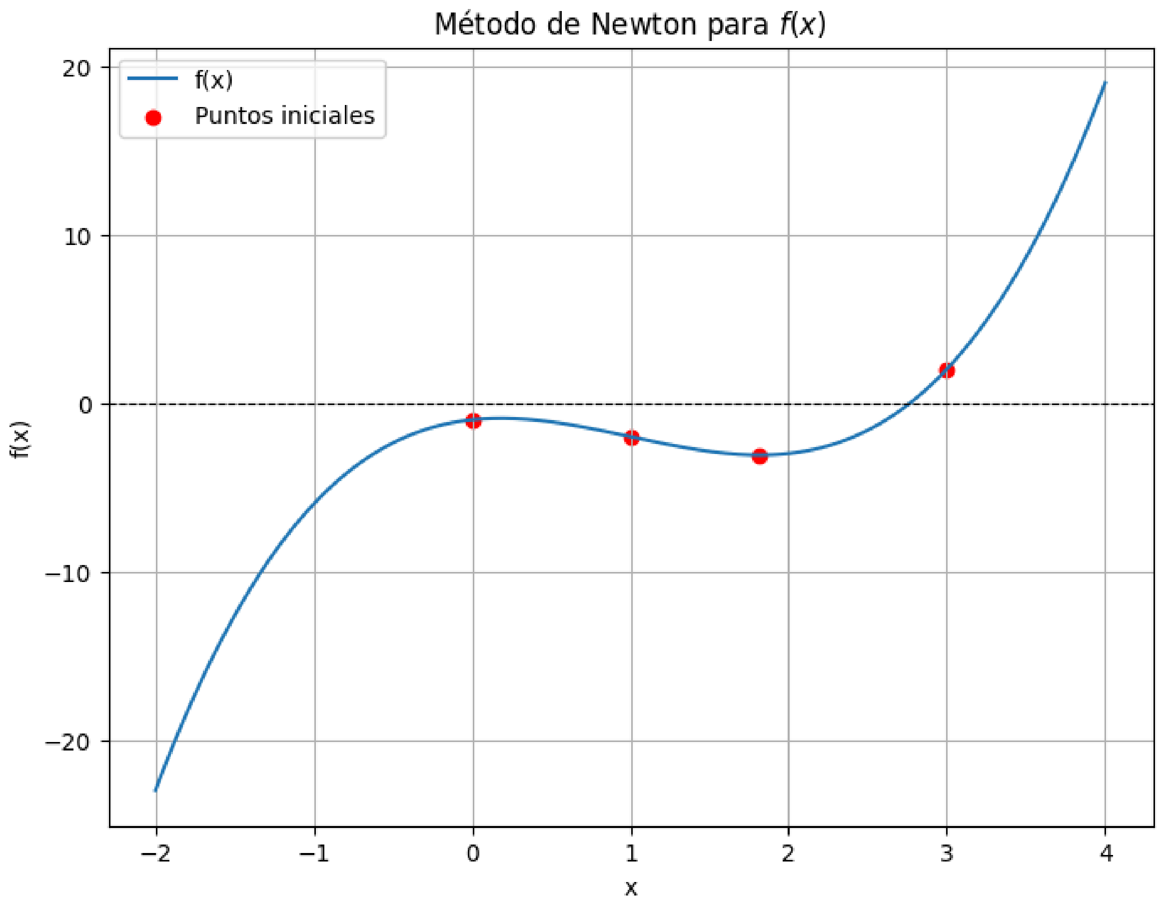
Código

In [ ]:

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

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

def f_prime(x):
    return 3*x**2 - 6*x + 1

# Método de Newton
def newton_method(x0, tol=1e-6, max_iter=100):
    x = x0
    for i in range(max_iter):
        if f_prime(x) == 0:
            return x, i, "Error: división por 0"
        x_new = x - f(x) / f_prime(x)
        if abs(x_new - x) < tol:
```

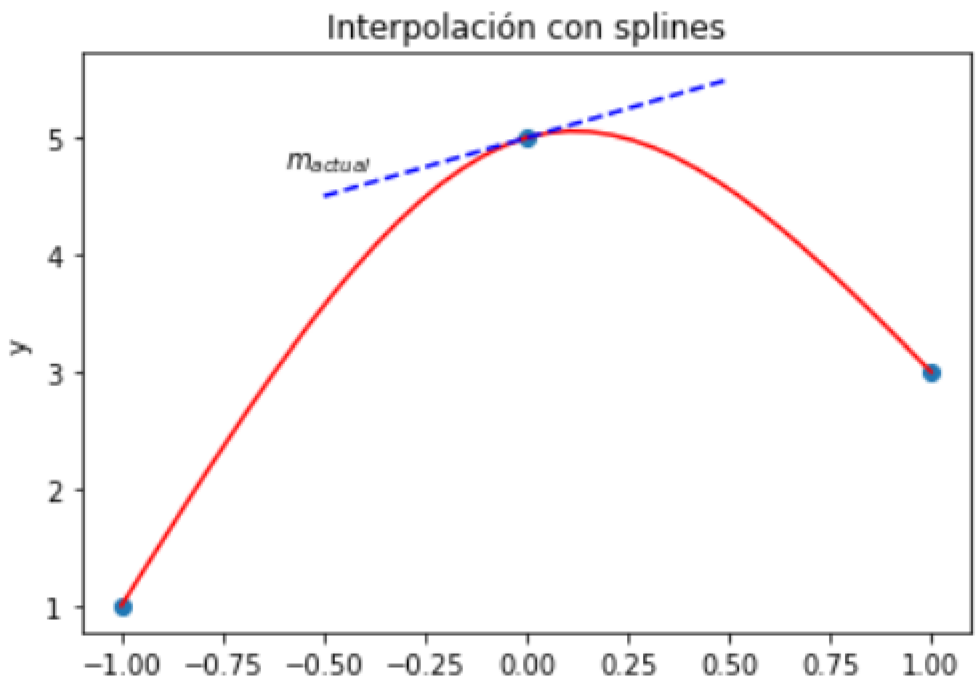
```python
            return x_new, i, "Convergencia"
        x = x_new
    return x, max_iter, "Error: no convergió"

# Evaluación de los casos
x_values = [3, 1, 0, 1 + np.sqrt(6)/3]
results = []

for x0 in x_values:
    root, iterations, status = newton_method(x0)
    results.append((x0, root, iterations, status))

# Resultados
for r in results:
    print(f"x0 = {r[0]:.6f} -> Raíz: {r[1]:.6f}, Iteraciones: {r[2]}, Estado: {r

# Gráfica
x = np.linspace(-2, 4, 500)
y = f(x)

plt.figure(figsize=(8, 6))
plt.plot(x, y, label="f(x)")
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.scatter([r[0] for r in results], [f(r[0]) for r in results], color='red', la
plt.title("Método de Newton para $f(x)$")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid()
plt.show()
```

```
x0 = 3.000000 -> Raíz: 2.769292, Iteraciones: 3, Estado: Convergencia
x0 = 1.000000 -> Raíz: 1.000000, Iteraciones: 100, Estado: Error: no convergió
x0 = 0.000000 -> Raíz: 0.000000, Iteraciones: 100, Estado: Error: no convergió
x0 = 1.816497 -> Raíz: 1.816497, Iteraciones: 0, Estado: Error: división por 0
```



Pregunta 6

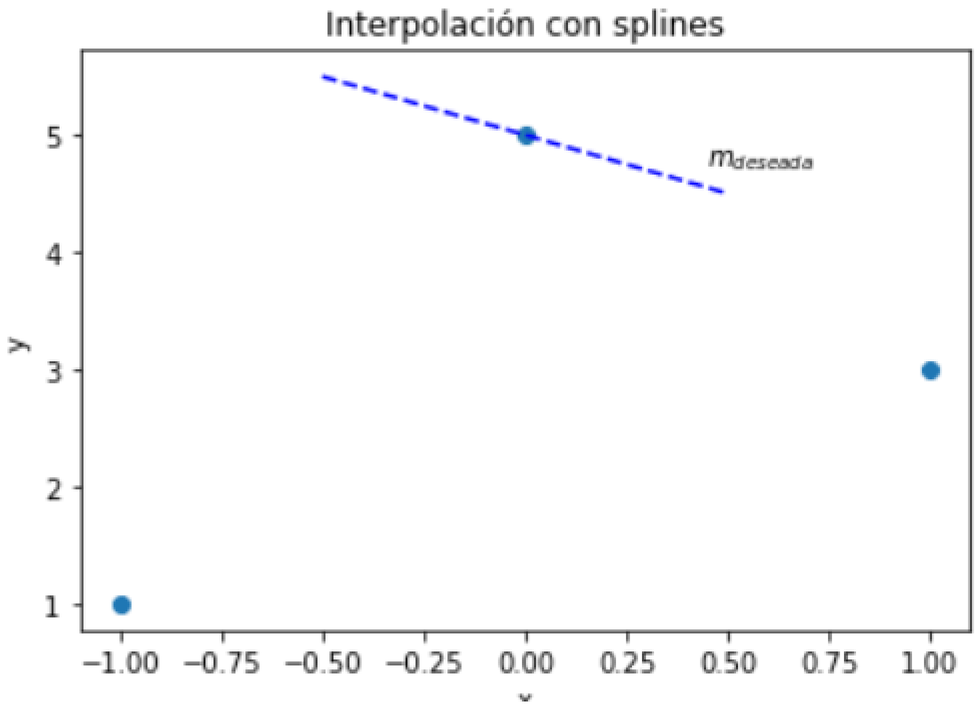
Dados los puntos $(-1, 1), (0, 5), (1, 3)$, se ha obtenido los splines cúbicos correspondientes.
Sin embargo, al observar la figura, usted no se siente satisfecho con la pendiente resultante en el punto (x_1, y_1). Y decide intentar una modificación a las ecuaciones, tal que los splines sean tangentes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



Recuerde que la expresión de un spline cúbico es la siguiente:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

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



Determine la ecuación que se debe modificar para poder cumplir con el requisito de m.

$S_0'(x_1) = S_1'(x_1)$

La expresión de un spline cúbico es:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Donde:

- $S_i(x_i) = y_i$ (el spline debe pasar por el punto inicial del intervalo),
- $S_i(x_{i+1}) = y_{i+1}$ (el spline debe pasar por el punto final del intervalo),
- $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$ (la derivada debe ser continua entre los intervalos),
- $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$ (la segunda derivada debe ser continua entre los intervalos).

En este caso, además de estas condiciones generales, debemos cumplir con una condición adicional que fija la pendiente $m = 2$ en el punto x_1.

Planteamos las ecuaciones para el spline S_0

Condiciones iniciales

- $S_0(-1) = y_0 = 1$:

$a_0 = 1$

- $S_0(0) = y_1 = 5$:

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

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

$$b_0 + c_0 + d_0 = 4 \quad (\text{Ecuación } 1).$$

- $S_0'(0) = S_1'(0)$ (continuidad de la derivada en $x_1 = 0$):

$$b_0 + 2c_0(1) + 3d_0(1)^2 = b_1$$

$$b_0 + 2c_0 + 3d_0 = b_1 \quad (\text{Ecuación } 2).$$

- $S_0''(0) = S_1''(0)$ (continuidad de la segunda derivada en $x_1 = 0$):

$$2c_0 + 6d_0 = 2c_1$$

$$c_0 + 3d_0 = c_1 \quad (\text{Ecuación } 3).$$

- Condición adicional: $S_0'(0) = m = 2$:

$$b_0 + 2c_0 + 3d_0 = 2 \quad (\text{Ecuación } 4).$$

Plantear las ecuaciones para el spline S_1

Condiciones iniciales

- $S_1(0) = y_1 = 5$:

$$a_1 = 5$$

- $S_1(1) = y_2 = 3$:

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

$$5 + b_1(1) + c_1(1)^2 + d_1(1)^3 = 3$$

$$b_1 + c_1 + d_1 = -2 \quad (\text{Ecuación } 5).$$

- **Condición adicional:** $S_1''(1) = 0$:

$$2c_1 + 6d_1 = 0$$

$$c_1 + 3d_1 = 0 \quad (\text{Ecuación } 6).$$

Resolvemos el sistema de ecuaciones Para (S_0): De la **Ecuación 1**:

$$b_0 + c_0 + d_0 = 4$$

De la **Ecuación 4**:

$$b_0 + 2c_0 + 3d_0 = 2$$

Restando las dos ecuaciones:

$(c_0 + 2d_0) = -2 \quad \text{(Ecuación 7)}.$

De la **Ecuación 3**:

$c_0 + 3d_0 = c_1$

Usando la **Ecuación 6** ($c_1 + 3d_1 = 0$):

$c_1 = -3d_1$

De la **Ecuación 7**:

$d_0 = -3, \quad c_0 = 0$

Finalmente, usando la **Ecuación 2**:

$b_0 + 2(-2) + 3(0) = 2$

$b_0 = 7$

Resolvemos para S_1: De la **Ecuación 5**:

$b_1 + c_1 + d_1 = -2$

De la **Ecuación 6**:

$c_1 = -3d_1$

Sustituyendo en la **Ecuación 5**:

$b_1 - 3d_1 + d_1 = -2$

$b_1 - 2d_1 = -2$

Dado que $b_1 = -2$ y $d_1 = 0$, tenemos:

$c_1 = 0$

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

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

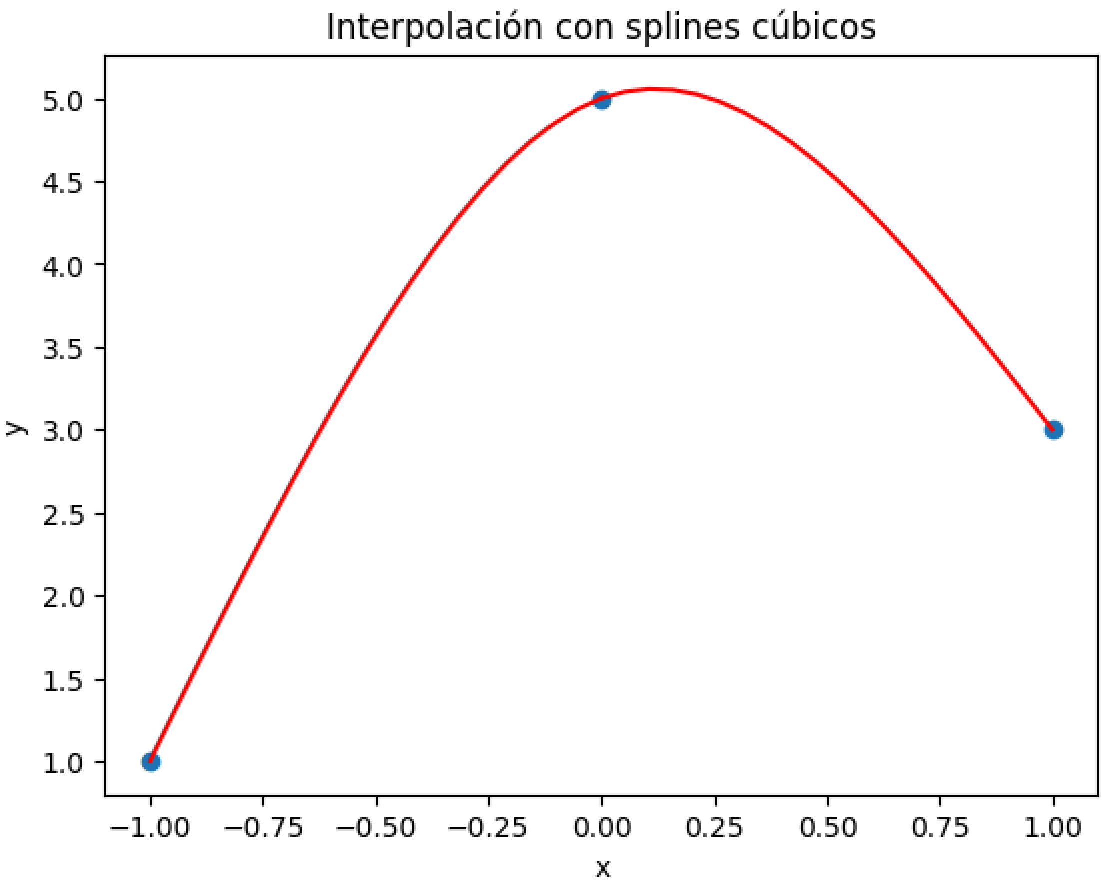
Escriba la expresión del spline S_1. En caso de no existir solución, llene los casilleros con Nan.

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

Para graficar su respuesta debe utilizar el código base del siguiente repositorio: https://github.com/ztjona/MN-examen-01-2024-B/blob/main/splines_pendiente.ipynb

Código proporcionado

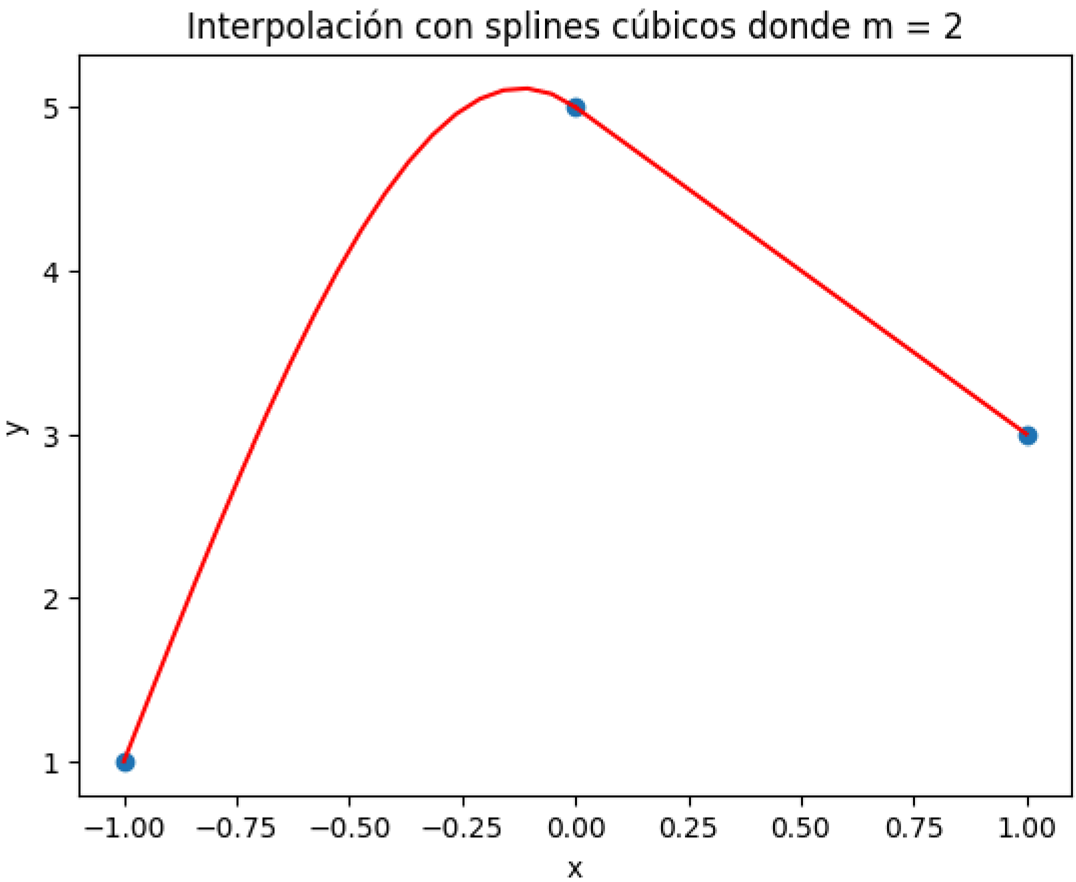
In [21]:
```python
def Spline(x: float, x0: float, pars: dict[str, float]) -> float:
    a = pars["a"]
    b = pars["b"]
    c = pars["c"]
    d = pars["d"]
    return a + b * (x - x0) + c * (x - x0) ** 2 + d * (x - x0) ** 3


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()
```



Gráfica de splines cubicos con pendiente m =2

In [23]:

```python
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


import matplotlib.pyplot as plt
import numpy as np

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

La interpolación de un conjunto de puntos usando polinomios de Lagrange $P(x)$ está dada por la fórmula:

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

Donde:

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

Dados los puntos $(0, 0), (1, 1), (2, 2), (3, 3)$.

Tenemos que:

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

$$L(x) = \frac{x\cdot(x-2)\cdot(x-3)}{1\cdot(-1)\cdot(-2)} + 2\frac{x\cdot(x-1)\cdot(x-3)}{2\cdot1\cdot(-1)} + 3\frac{x\cdot(x-1)\cdot(x-2)}{3\cdot2\cdot1}$$

$$L(x) = \frac{(x^2-2x)\cdot(x-3)}{(-1)\cdot(-2)} + 2\frac{(x^2-x)\cdot(x-3)}{2\cdot(-1)} + 3\frac{(x^2-x)\cdot(x-2)}{6\cdot1}$$

$$L(x) = \frac{(x^3-5x^2+6x)}{2} + 2\frac{(x^3-4x^2+3x)}{(-2)} + 3\frac{(x^3-3x^2+2x)}{6}$$

$$L(x) = \frac{1}{2}(x^3 - 5x^2 + 6x) - (x^3 - 4x^2 + 3x) + \frac{1}{2}(x^3 - 3x^2 + 2x)$$

$$L(x) = (\frac{1}{2}x^3 - \frac{5}{2}x^2 + 3x) + (-x^3 + 4x^2 - 3x) + (\frac{1}{2}x^3 - \frac{3}{2}x^2 + x)$$

$$L(x) = x$$

El polinomio resultante simplificado tiene orden $= 1$

Encuentre el polinomio de Lagrange respectivo, (simplifique la expresión para facilitar la evaluación) $P(x) = x$

Usando el polinomio $P(x)$ que obtuvo de respuesta, calcule:

$$P(x = 3.78) = 3.78$$

$$P(x = 19.102) = 19.102$$

Código

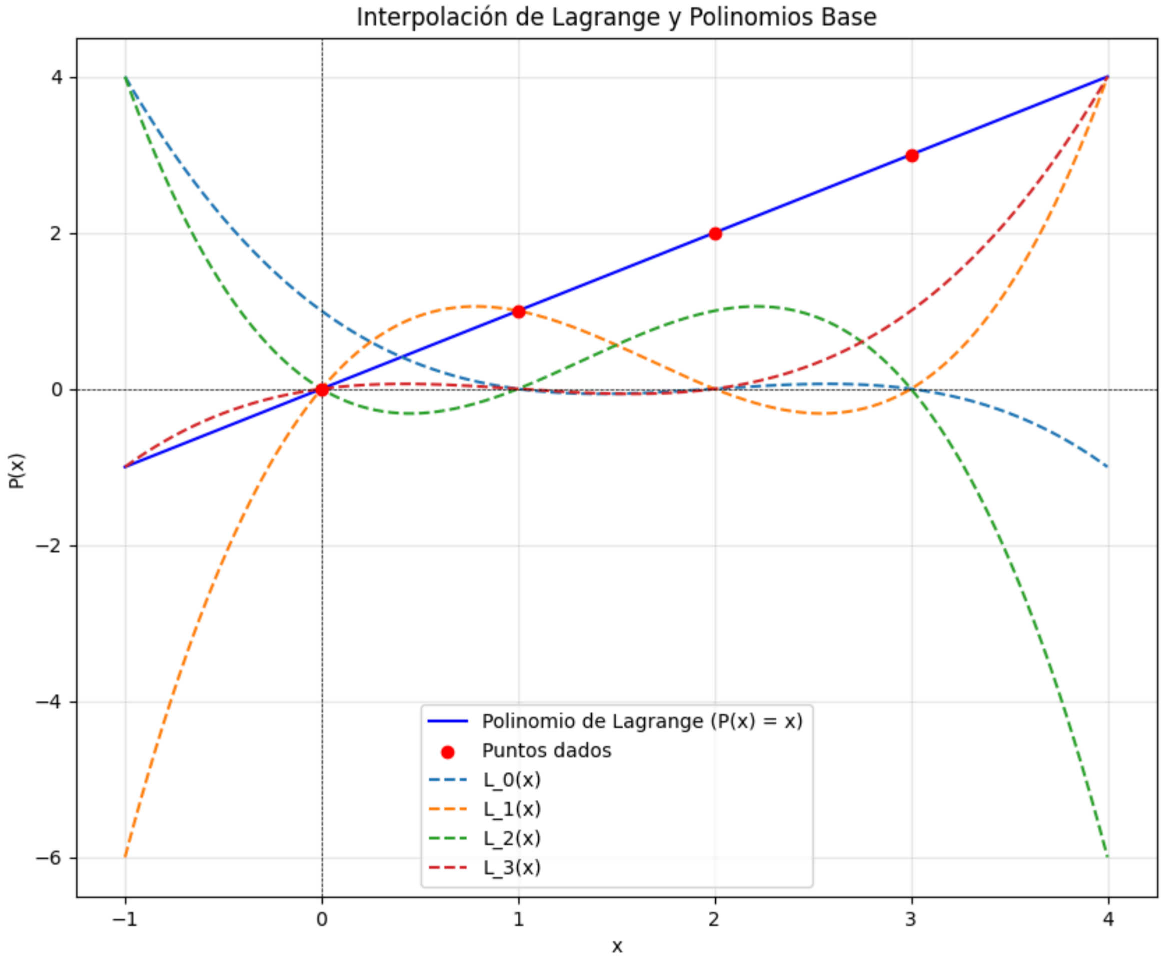
In [6]:

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

def lagrange_interpolation(x_points, y_points, x_eval):
    """
    Realiza la interpolación de Lagrange para los puntos dados y evalúa el polin

    Parameters:
    x_points: array-like, puntos x dados.
    y_points: array-like, puntos y dados.
```

```
    x_eval: float o array-like, puntos donde evaluar el polinomio.

    Returns:
    float o array-like, valores del polinomio evaluados en x_eval.
    """
    n = len(x_points)
    def L_k(k, x):
        term = 1
        for i in range(n):
            if i != k:
                term *= (x - x_points[i]) / (x_points[k] - x_points[i])
        return term

    def P(x):
        result = 0
        for k in range(n):
            result += y_points[k] * L_k(k, x)
        return result

    if isinstance(x_eval, (int, float)):
        return P(x_eval)
    else:
        return np.array([P(x) for x in x_eval])

# Datos de los puntos
x_points = [0, 1, 2, 3]
y_points = [0, 1, 2, 3]

# Evaluar el polinomio en los puntos pedidos
x_eval_1 = 3.78
x_eval_2 = 19.102
result_1 = lagrange_interpolation(x_points, y_points, x_eval_1)
result_2 = lagrange_interpolation(x_points, y_points, x_eval_2)

print(f"P(x=3.78) = {result_1}")
print(f"P(x=19.102) = {result_2}")

# Generar datos para la gráfica
x_plot = np.linspace(-1, 4, 500)
y_plot = lagrange_interpolation(x_points, y_points, x_plot)

# Generar los polinomios base L_k(x)
y_base = [np.array([np.prod([(x - x_points[j]) / (x_points[i] - x_points[j]) for

# Crear la gráfica
plt.figure(figsize=(10, 8))
plt.plot(x_plot, y_plot, label="Polinomio de Lagrange (P(x) = x)", color="blue")
plt.scatter(x_points, y_points, color="red", label="Puntos dados", zorder=5)

# Graficar los polinomios base
for i, y_lk in enumerate(y_base):
    plt.plot(x_plot, y_lk, linestyle="--", label=f"L_{i}(x)")

plt.title("Interpolación de Lagrange y Polinomios Base")
plt.xlabel("x")
plt.ylabel("P(x)")
plt.axhline(0, color="black", linewidth=0.5, linestyle="--")
plt.axvline(0, color="black", linewidth=0.5, linestyle="--")
plt.legend()
```

```
plt.grid(alpha=0.3)
plt.show()
```

```
P(x=3.78) = 3.780000000000001
P(x=19.102) = 19.10199999999986
```



Pregunta 8

Dados los puntos $(-1, 1), (1, 3)$. Determine el spline cúbico teniendo en cuenta que $f'(x_0) = 1$, $f'(x_n) = 2$.

Conocemos que un spline cúbico en un intervalo es:

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

donde:

- $x_0 = -1$
- $x_1 = 1$

tenemos cuatro coeficientes que los podemos determinar usando las siguientes condiciones:

1. $S(x_0) = y_0$

2. $S(x_1) = y_1$

3. $S'(x_0) = f'(x_0)$

4. $S'(x_1) = f'(x_1)$

Sustituimos:

condicion 1: $S(x_0) = y_0$

$S(-1) = a(-1+1)^3 + b(-1+1)^2 + c(-1_1) + d + 1$

$= 0 + 0 + 0 + d + 1 \Rightarrow d = 1$

Condición 2: $S(x_1) = y_1$

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

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

$8a + 4b = 2c = 2$ ecuación (1)

Condición 3: $S'(x_0) = f'(x_0)$

La derivada del spline es: $S'(x) = 3a(x-x_0)^2 + 2b(x-x_0) + c$

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

$0 + 0 + c = 1 \Rightarrow c = 1$

Condición 4: $S'(x_1) = f'(x_1)$

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

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

$12a + 4b = 1$ ecuación (2)

Resolvemos el sistema de ecuaciones: de (1) y (2)

restamos las ecuaciones: $(12a + 4b) - (8a + 4b) = 1 - 0$

$a = \frac{1}{4}$

sustituimos a en $8a + 4b = 0$

$8(\frac{1}{4}) + 4b = 0 \Rightarrow b = -\frac{1}{2}$

entonces tenemos que: $a = 0.25, b = -0.5, c = 1, d = 1$

sustituyendo los valores tenemos:

$S_0(x) = 0.25 * (x+1)^3 - 0.5 * (x+1)^2 + 1 * (x+1) + 1$

Código

In [7]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```python
# Definición del spline cúbico

def spline_cubico(x):
    """
    Calcula el valor del spline cúbico en el punto x.
    S(x) = 0.25 * (x + 1)^3 - 0.5 * (x + 1)^2 + 1 * (x + 1) + 1
    """
    return 0.25 * (x + 1)**3 - 0.5 * (x + 1)**2 + 1 * (x + 1) + 1

# Puntos dados
x_points = [-1, 1]
y_points = [1, 3]

# Valores de x para graficar
x_plot = np.linspace(-2, 2, 500)
y_plot = spline_cubico(x_plot)

# Crear la gráfica
plt.figure(figsize=(8, 6))
plt.plot(x_plot, y_plot, label="Spline Cúbico", color="blue")
plt.scatter(x_points, y_points, color="red", label="Puntos dados", zorder=5)
plt.title("Spline Cúbico para los puntos dados")
plt.xlabel("x")
plt.ylabel("S(x)")
plt.axhline(0, color="black", linewidth=0.5, linestyle="--")
plt.axvline(0, color="black", linewidth=0.5, linestyle="--")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```