Actividad evaluada #3

02MIAR

Matemáticas para la Inteligencia Artificial

Profesor

Dr. Matthieu F.-W. Huber

Grupo No. 11

Jonathan Mora
Luis Jama Tello
Blanca Santos Fernández
Laura Betancourt Leal

Junio 2024

Índice general

1 Ejercicios acerca del determinante

1.1 Desarrollo de Laplace.

1.1.1 Deducir de la definición 4 el determinante en dimensión 0, 1 y 2.

Dimensión 0 En dimensión 0, una matriz es simplemente un escalar, y su determinante es el propio escalar:

$$det(a) = a \tag{1}$$

Dimensión 1 Una matriz es un solo vector (o escalar). Para un vector $\mathbb{v}_{\mathbb{1}}$ en $\mathbb{R}$, el determinante es simplemente el valor absoluto del vector, ya que la única función lineal antisimétrica es la identidad.

$$\det([v_1]) = v_1 \tag{2}$$

Dimensión 2 En dimensión 2 tenemos una matriz 2x2. Sea $\mathbb{A}$ una matriz de $\mathbb{R}^2$:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{3}$$

La función lineal antisimetrica que cumple la definición 4 seria:

$$det(A) = ad - cd \tag{4}$$

La matriz identidad de una matriz 2x2 es:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5}$$

Por lo que tenemos que el resultado del determinate sera dado por:

$$det(A) = 1 \cdot 1 - 0 \cdot 0 = 1 \tag{6}$$

1.1.2 A partir de la definición 4, expresar el determinante de una matriz cuadrada recursivamente en función de determinantes la matrices cuadradas de dimensión inferior.

Indicación: para cada $n \in N$, distribuir (por linealidad en las columnas) sobre la descomposición

$$\begin{bmatrix} \lambda & \omega \\ v & A \end{bmatrix} = \begin{bmatrix} \lambda \cdot 1 + 0 & \omega \\ \lambda \cdot 0 + v & A \end{bmatrix}. \tag{7}$$

de una matriz cuadrada de dimensión + 1, siendo $n \in N$ y
- λ un coeficiente real,
- v un vector de dimensión (una columna de coeficientes reales),
- ω un covector de la misma dimensión (una fila de coeficientes),
- A una matriz cuadrada de la misma dimensión (con 2 coeficientes),
luego proceder del mismo modo con
- los demás coeficientes de esa primera columna,
- con cada columna.

Resolución ejercicio :
Diremos que n = 2 por lo que tendremos las siguientes igualdades :

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \tag{8}$$

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{9}$$

$$w = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \tag{10}$$

Usaremos una matriz M de dimensión 3×3 como ejemplo concreto donde estaran los valores λ, v, ω y los valores de A

$$M = \begin{bmatrix} \lambda & w_1 & w_2 \\ v_1 & a_{11} & a_{12} \\ v_2 & a_{21} & a_{22} \end{bmatrix} \tag{11}$$

Paso 1: Expansión por Cofactores

Para calcular el determinante de M, vamos a usar cofactores. Esto significa que vamos a descomponer el determinante de la matriz grande en términos de los determinantes de matrices más pequeñas.

$$det(M) = \lambda \cdot det\left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\right) - w_1 \cdot det\left(\begin{bmatrix} v_1 & a_{12} \\ v_2 & a_{22} \end{bmatrix}\right) + w_2 \cdot det\left(\begin{bmatrix} v_1 & a_{11} \\ v_2 & a_{21} \end{bmatrix}\right) \tag{12}$$

Entonces tenemos 3 sub matrices :

$$M_1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \tag{13}$$

$$M_2 = \begin{bmatrix} v_1 & a_{12} \\ v_2 & a_{22} \end{bmatrix} \tag{14}$$

$$M_3 = \begin{bmatrix} v_1 & a_{11} \\ v_2 & a_{21} \end{bmatrix} \tag{15}$$

Si simplificamos la ecuación tendríamos que:

$$det(M) = \lambda \cdot det(A) - w_1 \cdot det(M_2) + w_2 \cdot det(M_2) \tag{16}$$

Por tanto tenemos que en n:

$$M = \begin{pmatrix} \lambda & \omega_1 & \omega_2 & \cdots & \omega_n \\ v_1 & a_{11} & a_{12} & \cdots & a_{1n} \\ v_2 & a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_n & a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \tag{17}$$

El determinante esta dado por:

$$det(M) = \lambda \cdot det(A) - w_1 \cdot det(M_1) + w_2 \cdot det(M_2) ... + (-1)^{1+i} \cdot w_i \cdot det(A_i) \tag{18}$$

Que es similar a decir que:

$$\det \begin{pmatrix} \lambda & \omega \\ v & A \end{pmatrix} = \lambda \cdot \det(A) + \sum_{i=1}^{n} \omega_i \cdot (-1)^{1+i} \cdot \det(A_i) \tag{19}$$

1.1.3 Implementar en Python la definición así obtenida.

[1]:
```python
import numpy as np

def calcular_determinante_combinado(lambda_val, v, omega, A):
    # Convertir las entradas a matrices numpy
    v = np.array(v).reshape(-1, 1)
    omega = np.array(omega).reshape(1, -1)
    A = np.array(A)

    # Construir la matriz combinada
    primera_fila = np.hstack(([lambda_val], omega[0]))
    resto_filas = np.hstack((v, A))
    matriz_combinada = np.vstack((primera_fila, resto_filas))

    print("Matriz compuesta:")
    print(matriz_combinada)

    # Calcular el determinante de la matriz combinada
    determinante_combinado = np.linalg.det(matriz_combinada)

    # Verificación del determinante según la fórmula
    determinante_A = np.linalg.det(A)
    suma_terminos = 0
```

```
    cofactores = []
    for i in range(len(omega[0])):
        # Crear una submatriz A_i eliminando la primera fila y la columna i
        columna_a_eliminar = i + 1;
        num_filas, num_columnas = matriz_combinada.shape
        A_i = np.delete(matriz_combinada, 0, axis=0)  # Eliminar la primera fila
        A_i = np.delete(A_i, columna_a_eliminar, axis=1)  # Eliminar la columna
 ↪i
        #print(f"(A_{i+1}")
        #print(A_i)
        #print(f"w_{i+1}", omega[0][i])
        cofactor = omega[0][i] * (-1)**(1+i) * np.linalg.det(A_i)
        cofactores.append(cofactor)
        suma_terminos += cofactor
        #print(f"Cofactor {i+1}: omega[{i}] * (-1)^{1+i} * det(A_{i+1}) =
 ↪{omega[0][i]} * (-1)^{1+i} * {np.linalg.det(A_i)} = {cofactor}")

    determinante_verificado = lambda_val * determinante_A + suma_terminos

    # Mostrar cada uno de los cofactores y sus resultados
    print("Cofactores:")

    # Mostrar la fórmula completa en números
    formula = f"{lambda_val} * det(A) + " + " + ".join([f"{omega[0][i]} *
 ↪(-1)^{1+i} * det(A_{i+1})" for i in range(len(omega[0]))])
    formula_numerica = f"{lambda_val} * {determinante_A} + " + " + ".
 ↪join([str(cofactor) for cofactor in cofactores])

    print("Fórmula simbólica: ", formula)
    print("Fórmula numérica: ", formula_numerica)

    return matriz_combinada, determinante_combinado, determinante_verificado

# Ejemplo de uso
def generate_random_parameters(inf_lim, sup_lim):

    # 1. Generar n como un número entero aleatorio entre 2 y 5
    n = np.random.randint(2, 15)

    # 2. Generar lambda como un número entero aleatorio entre -10 y 10
    lambda_ = np.random.randint(inf_lim, sup_lim)

    # 3. Generar un vector columna v de dimensión n con valores enteros
    v = np.random.randint(inf_lim, sup_lim, n)

    # 4. Generar un covector omega de dimensión n con valores enteros
    omega = np.random.randint(inf_lim, sup_lim, n)
```

```
    # 5. Generar una matriz cuadrada A de dimensión n x n con valores enteros
    A = np.random.randint(inf_lim, sup_lim, (n, n))

    return n, lambda_, v, omega, A



n, lambda_, v, omega, A = generate_random_parameters(-11,15)

print("n:", n)
print("lambda:", lambda_)
print("v:", v)
print("omega:", omega)
print("A:")
print(A)

matriz_combinada, determinante_combinado, determinante_verificado =␣
 ↪calcular_determinante_combinado(lambda_, v, omega, A)

print("Determinante:", determinante_combinado)
print("Verificacion:", determinante_verificado)
```

```
n: 5
lambda: 0
v: [ 5 -9 -6  7  0]
omega: [-5  6 -6 11  1]
A:
[[  3   6  13   2  -9]
 [  1  -4   4   1  -5]
 [ -7   3   1   8   5]
 [ -2  14   5   1  -3]
 [ -2 -11   4  12  13]]
Matriz compuesta:
[[  0  -5   6  -6  11   1]
 [  5   3   6  13   2  -9]
 [ -9   1  -4   4   1  -5]
 [ -6  -7   3   1   8   5]
 [  7  -2  14   5   1  -3]
 [  0  -2 -11   4  12  13]]
Cofactores:
Fórmula simbólica:  0 * det(A) + -5 * (-1)^1 * det(A_1) + 6 * (-1)^2 * det(A_2)
+ -6 * (-1)^3 * det(A_3) + 11 * (-1)^4 * det(A_4) + 1 * (-1)^5 * det(A_5)
Fórmula numérica:  0 * 690.0000000000016 + 140600.00000000006 +
-46212.00000000012 + -131280.0000000005 + -336688.00000000093 +
10678.000000000053
Determinante: -362902.00000000076
```

```
Verificacion: -362902.00000000146
```

1.2 Ejercicio 2 : Eliminación de Gauss–Jordan.

1.2.1 Deducir de la definición 4 el efecto que tiene en el determinante de una matriz sumar a una de sus columnas una combinación lineal de las demás.

1.2.2 A partir de la definición 4, proponer una estrategia para triangularizar una matriz sin cambiar su determinante e implementar en Python una definición alternativa del determinante. Indicación: descomponer similarmente al ejercicio anterior.

1.2.3 Implementar en Python la definición así obtenida

[ ]:

1.3 Ejercicio 3: Comparación.

1.3.1 Obtener la complejidad computacional de cada una de estas dos implementaciones.

- Calcular complejidad computacional ejercicio: Determinante Matriz Combinada
- Función **generate_random_parameters(inf_lim, sup_lim)**:
  Tiene complejidad computacional **$O(n^2)$** dado que las líneas que generan vectores y matrices (puntos 3, 4 y 5) tienen complejidades O(n) y $O(n^2)$, respectivamente, y el resto de las líneas O(1), la complejidad total del código está dominada por la generación de la matriz 'A', es decir **$O(n^2)$**.
- Función **calcular_determinante_combinado(lambda_val, v, omega, A)**:
  La función tiene la siguiente estructura:
  - Construcción de la matriz combinada: Tiene un costo de O(n), donde n es el tamaño del vector v y la matriz A (asumiendo que omega tiene la misma longitud que v).
  - Cálculo del determinante de la matriz combinada: Utilizando np.linalg.det, el costo es O(n^3) en el peor de los casos para matrices cuadradas.
  - Cálculo de los cofactores: Se realiza un ciclo for de tamaño n (el tamaño de omega). Dentro del ciclo, se calcula el determinante de una submatriz de tamaño (n-1) x (n-1), con un costo de O((n-1)^3).

  Por lo tanto, el costo de calcular los cofactores es O(n * (n-1)^3), que es equivalente a O(n^4). Verificación del determinante: Tiene un costo de O(n), ya que se realiza una suma de n términos.

  "La complejidad computacional total del código es O(n^4), lo que significa que el tiempo de ejecución crece con la cuarta potencia del tamaño de la matriz. Este es un tiempo de ejecución relativamente alto, especialmente para matrices grandes."

1.3.2 Generar matrices aleatoriamente en dimensión $n \in \{ 2, 3, \cdot \cdot , 9, 10 \}$ y comparar el tiempo de ejecución de cada una de estas dos implementaciones con la función numpy.linalg.det (la función determinante de la extensión numérica de Python al álgebra lineal). Indicación: se puede utilizar la función numpy.random.rand para generar los coeficientes aleatorios de sus matrices.

[2]:
```
import numpy as np
import timeit
```

[3]:
```
def det_laplace(matriz: np.ndarray) -> float:
  """Calcula el determinante de una matriz usando la regla de Laplace.

    Parameters
    ----------
        matriz: Es la matriz n x n para la cual se calcula el determinante

    Returns
    -------
    float
        El determinante de la matriz
  """

  n = len(matriz)

  if n == 1:
    return matriz[0][0]
  elif n == 2:
    return matriz[0][0] * matriz[1][1] - matriz[0][1] * matriz[1][0]
  else:
    det = 0
    for j in range(n):
      submatriz = np.delete(np.delete(matriz, 0, axis=0), j, axis=1)
      det += (-1) ** j * matriz[0][j] * det_laplace(submatriz)
    return det


def det_gauss_jordan(matriz: np.ndarray) -> float:
  """Calcula el determinante de una matriz usando el método de eliminación de␣
 ↪Gauss-Jordan.

    Parameters
    ----------
        matriz: Es la matriz n x n para la cual se calcula el determinante

    Returns
    -------
    float
```

```
        El determinante de la matriz
    """

    n = len(matriz)
    det = 1

    for i in range(n):
        pivot = matriz[i][i]
        if pivot == 0:
            return 0

        det  *= pivot
        matriz[i] /= pivot

        for j in range(i+1, n):
            factor = matriz[j][i]
            matriz[j]  -= factor * matriz[i]
    return det
```

[4]:
```
print("Dimensión".ljust(10) + "| T.Laplace (ms)".ljust(17) + "| T.Gauss-Jordan (ms)".ljust(22) + "| T.numpy.linalg.det (ms)")
print("-" * 75)

# Genera matrices aleatorias de tamaño n x n
for n in range(2, 11):
    A = np.random.rand(n, n)

    # Calcula el tiempo de ejecución para cada algoritmo
    laplace_time = timeit.timeit(lambda: det_laplace(A), number=1) * 1000
    gauss_jordan_time = timeit.timeit(lambda: det_gauss_jordan(A), number=1)  * 1000
    numpy_time = timeit.timeit(lambda: np.linalg.det(A), number=1)  * 1000

    print(f"{str(n).ljust(9)} | {format(laplace_time, '.6f').ljust(14)} | {format(gauss_jordan_time, '.6f').ljust(19)} | {format(numpy_time, '.6f')}")
```

```
Dimensión | T.Laplace (ms) | T.Gauss-Jordan (ms) | T.numpy.linalg.det (ms)
---------------------------------------------------------------------------
2         | 0.009608       | 0.046513            | 0.035703
3         | 0.410473       | 0.042401            | 0.024766
4         | 0.830461       | 0.063236            | 0.026046
5         | 2.359853       | 0.090583            | 0.040796
6         | 13.553588      | 0.130150            | 0.036440
7         | 80.875877      | 0.162143            | 0.046795
8         | 643.808716     | 0.210978            | 0.045501
```

```
9          | 5897.031685     | 0.237919              | 0.040963
10         | 58559.983033    | 0.291078              | 0.039791
```

2 Ejercicios acerca del gradiente

2.1 Ejercicio 4 : Método descenso del gradiente

Con el propósito de aproximar un mínimo local de una función real de varias variables reales, el método de descenso de gradiente consiste en iterar una marcha (positivamente) proporcional al (opuesto del) gradiente desde un valor inicial, con la intuición de 'seguir el agua' hasta dar con el valle.

2.1.1 Implementar en Python un algoritmo de descenso del gradiente (con un máximo de m = 10^5 iteraciones) a partir de los siguientes argumentos tomados en ese orden:

- la función f cuyo mínimo local se propone aproximar,
- el valor inicial x desde el que empieza la marcha,
- la razón geométrica o coeficiente de proporcionalidad y,
- el parámetro de tolerancia z para finalizar cuando el gradiente de la función f caiga dentro de esa tolerancia.

Indicación: empezar por implementar el gradiente grad(f) de la función f.

[5]:
```python
import numpy as np

def gradiente(f, x, h=1e-8):
    """
    Aproximación del gradiente de f en el punto x usando diferencias finitas.

    f: La función de la cual se va a calcular el gradiente.
    x: El punto en el cual se va a calcular el gradiente.
    h: Un pequeño incremento para calcular las diferencias finitas.
    return: El gradiente de f en x.
    """
    n = len(x)
    gradiente = np.zeros(n)
    for i in range(n):
        x0 = np.copy(x)
        x1 = np.copy(x)
        x1[i] += h
```

```
        gradiente[i] = (f(x1) - f(x0)) / h
    return gradiente

def descenso_gradiente(f, x0, tasa_aprendizaje, tolerancia, max_iter=10**5):
    """
    Algoritmo de descenso de gradiente para encontrar el mínimo de una función␣
 ↪f.

    :param f: La función cuyo mínimo local se desea encontrar.
    :param x0: El punto inicial desde donde comienza la búsqueda.
    :param tasa_aprendizaje: La razón geométrica o coeficiente de␣
 ↪proporcionalidad.
    :param tolerancia: El parámetro de tolerancia para finalizar cuando el␣
 ↪gradiente de f esté dentro de esa tolerancia.
    :param max_iter: El número máximo de iteraciones.
    :return: El punto que minimiza la función f.
    """
    x = x0
    for i in range(max_iter):
        grad = gradiente(f, x)
        if np.linalg.norm(grad) < tolerancia:
            print(f"Convergió después de {i} iteraciones.")
            return x
        x = x - tasa_aprendizaje * grad
    print(f"Alcanzó el máximo de iteraciones ({max_iter}) sin convergencia.")
    return x
```

[6]:
```
# Ejemplo de uso:
# Definimos una función cuadrática simple para demostrar el descenso de␣
 ↪gradiente.
def f(x):
    return x[0]**2 + x[1]**2

# Punto inicial
x0 = np.array([10, 10])
# Razón geométrica o coeficiente de proporcionalidad
tasa_aprendizaje = 0.1
# Tolerancia
tolerancia = 1e-6

# Ejecutar el descenso de gradiente
punto_minimo = descenso_gradiente(f, x0, tasa_aprendizaje, tolerancia)
print("Punto mínimo encontrado:", punto_minimo)
```

```
Convergió después de 0 iteraciones.
Punto mínimo encontrado: [10 10]
```

2.1.2 Calcular formalmente $\{t \in R. f'(t) = 0\}$ para $f : t \mapsto 3t^4 + 4t^3 - 12t^2 + 7$.

Para resolver este problema, primero necesitamos calcular la derivada de la función ($f(t) = 3t^4 + 4t^3 - 12t^2 + 7$) y luego encontrar los puntos donde la derivada es igual a cero. Estos puntos son los candidatos para los mínimos y máximos locales de la función.

Vamos a proceder paso a paso:

1. Calcular la derivada de (f(t)).
2. Encontrar los puntos donde la derivada es igual a cero.
3. Utilizar la función de descenso de gradiente modificada para encontrar estos puntos.

- Calcular la derivada de $(f(t))$

La derivada de $(f(t))$ es:

$[f'(t) = \frac{d}{dt}(3t^4 + 4t^3 - 12t^2 + 7) = 12t^3 + 12t^2 - 24t]$

- Encontrar los puntos donde la derivada es igual a cero

Queremos encontrar los puntos (t) donde $(f'(t) = 0)$. Esto se traduce en resolver la ecuación:

$[12t^3 + 12t^2 - 24t = 0]$

Podemos factorizar esta ecuación:

$[12t(t^2 + t - 2) = 0]$

Esta factorización da tres posibles soluciones:

$[12t = 0 \quad \text{o} \quad t^2 + t - 2 = 0]$

Resolviendo estas ecuaciones:

1. $(12t = 0)$ nos da $(t = 0)$.
2. $(t^2 + t - 2 = 0)$ se puede resolver usando la fórmula cuadrática:

$[t = \frac{-1 \pm \sqrt{1^2 - 4 \cdot 1 \cdot (-2)}}{2 \cdot 1} = \frac{-1 \pm \sqrt{1+8}}{2} = \frac{-1 \pm 3}{2}]$

Esto nos da dos soluciones:

$[t = 1 \quad \text{y} \quad t = -2]$

Por lo tanto, los puntos donde $(f'(t) = 0)$ son $(t = 0), (t = 1)$, y $(t = -2)$.

- Implementación en Python de la función descenso de gradiente modificada para encontrar estos puntos

[7]:
```
import numpy as np

# Derivada de la función f(t)
def f_prima(t):
    return 12*t**3 + 12*t**2 - 24*t

# Descenso de gradiente adaptado para encontrar raíces de la derivada
def descenso_gradiente_raices(f_prima, x0, tasa_aprendizaje, tolerancia,␣
 ↪max_iter=10**5):
```

```
    x = x0
    for i in range(max_iter):
        gradiente = f_prima(x)
        if abs(gradiente) < tolerancia:
            print(f"Convergió después de {i} iteraciones.")
            return x
        x = x - tasa_aprendizaje * gradiente
    print(f"Alcanzó el máximo de iteraciones ({max_iter}) sin convergencia.")
    return x

# Parámetros para el descenso de gradiente
tasa_aprendizaje = 0.01
tolerancia = 1e-6

# Puntos iniciales para encontrar las raíces de la derivada
puntos_iniciales = [0, 1, -2]

# Ejecutar el descenso de gradiente para cada punto inicial
raices = []
for x0 in puntos_iniciales:
    raiz = descenso_gradiente_raices(f_prima, x0, tasa_aprendizaje, tolerancia)
    raices.append(raiz)
    print(f"Raíz encontrada: {raiz}")

print("Todas las raíces:", raices)
```

```
Convergió después de 0 iteraciones.
Raíz encontrada: 0
Convergió después de 0 iteraciones.
Raíz encontrada: 1
Convergió después de 0 iteraciones.
Raíz encontrada: -2
Todas las raíces: [0, 1, -2]
```

1. **Función `f_prima`**: Calcula la derivada de (f(t)).
2. **Función `descenso_gradiente_raices`**: Aplica el descenso de gradiente para encontrar las raíces de la derivada de la función.
3. **Parámetros para el descenso de gradiente**: Definimos la tasa de aprendizaje y la tolerancia.
4. **Puntos iniciales**: Usamos los puntos (t = 0), (t = 1), y (t = -2) como puntos de partida para verificar las raíces.
5. **Ejecutar el descenso de gradiente**: Para cada punto inicial, ejecutamos el descenso de gradiente y almacenamos los resultados.

2.1.3 Con una tolerancia $z = 10^{-12}$ y un valor inicial de $x = 3$ aplicar su algoritmo con razón $y = 10^{-1}$, 10^{-2}, 10^{-3} luego hacer lo mismo con $x = 0$. Interpretar el resultado.

Para abordar este ejercicio, vamos a aplicar el algoritmo de descenso de gradiente a la función $(f(t) = 3t^4 + 4t^3 - 12t^2 + 7)$ utilizando diferentes tasas de aprendizaje (learning rates) y dos puntos iniciales: $(x = 3)$ y $(x = 0)$. Utilizaremos una tolerancia de $(z = 10^{-12})$.

Primero, recordemos la derivada de la función:

$[f'(t) = 12t^3 + 12t^2 - 24t.]$

2.1.3.1 Implementación del Algoritmo

Vamos a implementar el descenso de gradiente para encontrar las raíces de $(f'(t))$ utilizando las tasas de aprendizaje dadas. Luego, interpretaremos los resultados para $(x = 3)$ y $(x = 0)$.

[8]:
```
import numpy as np

# Derivada de la función f(t)
def f_prime(t):
    return 12*t**3 + 12*t**2 - 24*t

# Descenso de gradiente adaptado para encontrar raíces de la derivada
def descenso_gradiente_raices(f_prime, x0, tasa_aprendizaje, tolerancia,␣
 ↪max_iter=10**5):
    x = x0
    for i in range(max_iter):
        gradiente = f_prime(x)
        if abs(gradiente) < tolerancia:
            print(f"Convergió después de {i} iteraciones.")
            return x
        x = x - tasa_aprendizaje * gradiente

        # Verificación de límites para prevenir overflow
        if abs(x) > 1e10:
            print(f"El valor de x = {x} se volvió demasiado grande en la␣
 ↪iteración {i}.")
            return None

    print(f"Alcanzó el máximo de iteraciones ({max_iter}) sin convergencia.")
    return x
```

2.1.3.2 Aplicación del Algoritmo

- **Valor inicial** $(x = 3)$
- **Tasa de aprendizaje** $(y = 10^{-1})$

[9]:
```
# Parámetros
tolerancia = 1e-12
```

```
punto_inicial = 3
tasa_aprendizaje = 0.1

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(f_prime, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
if raiz is not None:
    print(f"Raíz encontrada: {raiz}\n")
else:
    print("No se encontró una raíz dentro de los límites permitidos.")
```

```
Comenzando en x = 3 con tasa de aprendizaje = 0.1
El valor de x = -87049951065956.78 se volvió demasiado grande en la iteración 2.
No se encontró una raíz dentro de los límites permitidos.
```

Interpretación:

La tasa de aprendizaje ($y = 10^{-1}$) es demasiado alta. Esto hace que las actualizaciones en $ x $ sean muy grandes, causando que los valores se vuelvan extremadamente grandes en poco tiempo, lo que lleva a un desbordamiento numérico. Este es un claro ejemplo de cómo una tasa de aprendizaje demasiado grande puede desestabilizar el algoritmo de descenso de gradiente.

- **Valor inicial** ($x = 3$)
- **Tasa de aprendizaje** $y = 10^{-2}$

[10]:
```
# Parámetros
tasa_aprendizaje = 0.01

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(f_prime, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = 3 con tasa de aprendizaje = 0.01
Convergió después de 31 iteraciones.
Raíz encontrada: -1.9999999999999882
```

Interpretación:

Con una tasa de aprendizaje de ($y = 10^{-2}$), el algoritmo es más estable y converge rápidamente a una raíz cercana a -2. Este valor es uno de los puntos donde la derivada de la función original es cero, lo que indica un mínimo o máximo local.

- **Valor inicial** ($x = 3$)
- **Tasa de aprendizaje** $y = 10^{-3}$

[11]:
```
# Parámetros
tasa_aprendizaje = 0.001

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(f_prime, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = 3 con tasa de aprendizaje = 0.001
Convergió después de 831 iteraciones.
Raíz encontrada: 1.0000000000000275
```

Interpretación:

Con una tasa de aprendizaje aún más pequeña, ($y = 10^{-3}$), el algoritmo converge de manera más lenta (requiriendo 831 iteraciones). Sin embargo, alcanza un punto cercano a 1, que es otro punto donde la derivada de la función original es cero. Esto demuestra que una tasa de aprendizaje más pequeña puede llevar a una mayor precisión, aunque a costa de más iteraciones.

- **Valor inicial** $x = 0$
- **Tasa de aprendizaje** $y = 10^{-2}$

[12]:
```
# Parámetros
punto_inicial = 0
tasa_aprendizaje = 0.1

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(f_prime, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = 0 con tasa de aprendizaje = 0.1
Convergió después de 0 iteraciones.
Raíz encontrada: 0
```

Interpretación:

Al comenzar en ($x = 0$), el valor inicial ya es un punto donde la derivada de la función es cero. No se necesitan iteraciones adicionales porque 0 es una raíz de la derivada. Esto muestra que el algoritmo detecta correctamente que ya está en un punto estacionario.

Tasa de aprendizaje $y = 10^{-1}$

[13]:
```
# Parámetros
tasa_aprendizaje = 0.01
```

```
print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
  ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(f_prime, punto_inicial, tasa_aprendizaje,␣
  ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = 0 con tasa de aprendizaje = 0.01
Convergió después de 0 iteraciones.
Raíz encontrada: 0
```

Interpretación:

Igual que con ($y = 10^{-1}$), el algoritmo reconoce que el punto inicial $x = 0 $ ya es una raíz de la derivada. No se requieren actualizaciones adicionales.

Tasa de aprendizaje $y = 10^{-3}$

[14]:
```
# Parámetros
tasa_aprendizaje = 0.001

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
  ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(f_prime, punto_inicial, tasa_aprendizaje,␣
  ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = 0 con tasa de aprendizaje = 0.001
Convergió después de 0 iteraciones.
Raíz encontrada: 0
```

Interpretación:

Nuevamente, al comenzar en $x = 0$, el valor inicial ya es una raíz de la derivada. La tasa de aprendizaje no afecta el resultado en este caso porque no se necesitan iteraciones adicionales.

2.1.3.3 Conclusión General

- **Tasa de aprendizaje grande**: Puede llevar a desbordamientos numéricos o a oscilaciones alrededor de la raíz, como se observa con ($y = 10^{-1}$) cuando el valor inicial es ($x = 3$).
- **Tasa de aprendizaje moderada**: Proporciona un equilibrio entre velocidad y estabilidad, permitiendo una convergencia rápida y precisa, como se observa con ($y = 10^{-2}$) cuando el valor inicial es ($x = 3$).
- **Tasa de aprendizaje pequeña**: Garantiza una alta precisión, aunque a costa de un mayor número de iteraciones, como se observa con ($y = 10^{-3}$) cuando el valor inicial es ($x = 3$).
- **Valor inicial en la raíz**: Si el valor inicial es ya una raíz como ($x = 0$), el algoritmo converge instantáneamente sin necesidad de iteraciones adicionales.

Estos resultados ilustran cómo la elección de la tasa de aprendizaje y el punto inicial pueden influir significativamente en el comportamiento y eficiencia del algoritmo de descenso de gradiente.

2.1.4 Repetir estos dos últimos apartados con : (,) ² + 3 + ³ + 1 y los valores iniciales x = [-1,1], [0,0].

2.1.4.1: Calcular formalmente $(s,t) \in R^2 | f(s,t) = 0$

La función dada es: $[f(s,t) = s^2 + 3st + t^3 + 1]$

El gradiente de $(f(s,t))$ es: $[\nabla f(s,t) = \left(\frac{\partial f}{\partial s}, \frac{\partial f}{\partial t}\right)]$

Calculando las derivadas parciales:

$[\frac{\partial f}{\partial s} = 2s + 3t]$ $[\frac{\partial f}{\partial t} = 3s + 3t^2]$

Queremos encontrar los puntos donde el gradiente es cero:

$[2s + 3t = 0]$

$[3s + 3t^2 = 0]$

Resolviendo estas ecuaciones simultáneamente:

1. De la primera ecuación: $[2s + 3t = 0 \implies s = -\frac{3}{2}t]$
2. Sustituyendo $(s = -\frac{3}{2}t)$ en la segunda ecuación:

- $[3\left(-\frac{3}{2}t\right) + 3t^2 = 0]$
- $[-\frac{9}{2}t + 3t^2 = 0]$
- $[t(3t - \frac{9}{2}) = 0]$
- $[t = 0 \text{ o } t = \frac{3}{2}]$

Para $(t = 0)$: $[s = 0]$

Para $(t = \frac{3}{2})$: $[s = -\frac{3}{2}\left(\frac{3}{2}\right) = -\frac{9}{4}]$

Entonces, los puntos críticos son: $[(0,0)]$ $[(-\frac{9}{4}, \frac{3}{2})]$

2.1.4.2 Aplicar el algoritmo de descenso de gradiente

[15]:
```
import numpy as np

# Gradiente de la función f(s, t)
def gradiente_f(st):
    s, t = st
    df_ds = 2 * s + 3 * t
    df_dt = 3 * s + 3 * t**2
    return np.array([df_ds, df_dt])

# Descenso de gradiente adaptado para encontrar raíces del gradiente
def descenso_gradiente_raices(grad_f, x0, tasa_aprendizaje, tolerancia,␣
 ↪max_iter=10**5):
    x = np.array(x0, dtype=float)
    for i in range(max_iter):
        grad = grad_f(x)
        if np.linalg.norm(grad) < tolerancia:
            print(f"Convergió después de {i} iteraciones.")
```

```
            return x
        x = x - tasa_aprendizaje * grad
    print(f"Alcanzó el máximo de iteraciones ({max_iter}) sin convergencia.")
    return x
```

- **Valor inicial** $[-1, 1]$
- **Tasa de aprendizaje** $(y = 10^{-1})$

[16]:
```
# Parámetros
tolerancia = 1e-12
punto_inicial = [-1, 1]
tasa_aprendizaje = 0.1

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(gradiente_f, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = [-1, 1] con tasa de aprendizaje = 0.1
Convergió después de 302 iteraciones.
Raíz encontrada: [-2.25  1.5 ]
```

Interpretación:

La tasa de aprendizaje de 0.1 es moderada, permitiendo que el algoritmo converja razonablemente rápido a la raíz ($[-2.25, 1.5]$). Este punto es una de las soluciones donde el gradiente de la función es cero. El número de iteraciones es relativamente bajo, lo que indica una buena convergencia con esta tasa de aprendizaje.

- **Valor inicial** $[-1, 1]$
- **Tasa de aprendizaje** $(y = 10^{-2})$

[17]:
```
# Parámetros
tasa_aprendizaje = 0.01

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(gradiente_f, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = [-1, 1] con tasa de aprendizaje = 0.01
Convergió después de 3139 iteraciones.
Raíz encontrada: [-2.25  1.5 ]
```

Interpretación:

Con una tasa de aprendizaje de 0.01, el algoritmo converge más lentamente que con una tasa de 0.1, pero aún llega a la misma raíz ($[-2.25, 1.5]$). El número de iteraciones es significativamente mayor debido a la menor tasa de aprendizaje, lo que resulta en pasos más pequeños hacia la convergencia.

- **Valor inicial** $[-1, 1]$
- **Tasa de aprendizaje** $(y = 10^{-3})$

[18]:
```
# Parámetros
tasa_aprendizaje = 0.001

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(gradiente_f, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = [-1, 1] con tasa de aprendizaje = 0.001
Convergió después de 31558 iteraciones.
Raíz encontrada: [-2.25  1.5 ]
```

Interpretación: Con una tasa de aprendizaje aún más pequeña de 0.001, el algoritmo requiere muchas más iteraciones $(31,559)$ para converger a la misma raíz ($[-2.25, 1.5]$). Esto demuestra que una tasa de aprendizaje muy baja resulta en una convergencia muy lenta, aunque sigue siendo precisa.

- **Valor inicial** $([0, 0])$
- **Tasa de aprendizaje** $(y = 10^{-1})$

[19]:
```
# Parámetros
punto_inicial = [0, 0]
tasa_aprendizaje = 0.1

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(gradiente_f, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = [0, 0] con tasa de aprendizaje = 0.1
Convergió después de 0 iteraciones.
Raíz encontrada: [0. 0.]
```

Interpretación:

El valor inicial $([0, 0])$ ya es un punto donde el gradiente de la función es cero. Por lo tanto, el algoritmo no necesita realizar ninguna iteración adicional para encontrar la raíz. Esto muestra que el punto inicial ya es una solución, independientemente de la tasa de aprendizaje.

- **Valor inicial** $([0, 0])$

- **Tasa de aprendizaje** ($y = 10^{-2}$)

[20]:
```
# Parámetros
tasa_aprendizaje = 0.01

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(gradiente_f, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = [0, 0] con tasa de aprendizaje = 0.01
Convergió después de 0 iteraciones.
Raíz encontrada: [0. 0.]
```

Interpretación:

Igual que con la tasa de 0.1, el algoritmo reconoce que el punto inicial $[0, 0]$) ya es una raíz de la función, por lo que no se requieren iteraciones adicionales.

- **Valor inicial** ($[0, 0]$)
- **Tasa de aprendizaje** ($y = 10^{-3}$)

[21]:
```
# Parámetros
tasa_aprendizaje = 0.001

print(f"Comenzando en x = {punto_inicial} con tasa de aprendizaje =␣
 ↪{tasa_aprendizaje}")
raiz = descenso_gradiente_raices(gradiente_f, punto_inicial, tasa_aprendizaje,␣
 ↪tolerancia)
print(f"Raíz encontrada: {raiz}\n")
```

```
Comenzando en x = [0, 0] con tasa de aprendizaje = 0.001
Convergió después de 0 iteraciones.
Raíz encontrada: [0. 0.]
```

Interpretación:

Al igual que con las tasas de aprendizaje mayores, el punto inicial ($[0, 0]$) ya es una raíz, y el algoritmo no necesita realizar iteraciones adicionales.

2.1.4.3 Conclusión General

- **Tasa de aprendizaje y valor inicial** $[-1, 1]$:
- Una tasa de aprendizaje mayor (0.1) permite una convergencia más rápida con menos iteraciones.
- Una tasa de aprendizaje moderada (0.01) resulta en una convergencia más lenta pero estable.

- Una tasa de aprendizaje muy pequeña (0.001) lleva a una convergencia extremadamente lenta, aunque precisa.
- Todos convergen a la misma raíz ([-2.25, 1.5]), que es un punto crítico de la función.
- **Tasa de aprendizaje y valor inicial** $[0, 0]$:
- El valor inicial ([0, 0]) ya es un punto crítico donde el gradiente es cero.
- Independientemente de la tasa de aprendizaje, el algoritmo reconoce inmediatamente que está en la raíz y no realiza iteraciones adicionales.
- Esto muestra que cuando el punto inicial es ya un punto crítico, la tasa de aprendizaje no influye en el resultado.

Estos resultados ilustran cómo la elección de la tasa de aprendizaje afecta la velocidad de convergencia del algoritmo de descenso de gradiente y cómo los puntos críticos iniciales pueden simplificar la convergencia.