Matemáticas para la Inteligencia Artificial - Actividad Evaluada

Nombre: *__Josseph Yaakob Catagua Cobos__*

https://github.com/HikariJY/02MIAR_04_A_2024-25_Matematicas-para-la-Inteligencia-Artificial

Ejercicios Evaluados

In [1]:
```
import numpy as np
import math
import itertools
import time
import pandas as pd
import sympy as sp
```

In [2]:
```
def matriz_entera_cuadrada_random(min_value:int=-10, max_value:int=10, size_matrix:
    if size_matrix < 1:
        raise ValueError('La matriz no puede tener 0 elementos o menos.')
    elif type_matrix > 3 and type_matrix < 0:
        raise ValueError('El tipo de matriz debe ser:\n0) Matriz completa.\n1) Matr
    else:
        matrix = np.random.randint(min_value, max_value, size=(size_matrix, size_ma
        if type_matrix == 1:
            matrix = np.diag(np.diag(matrix)) # Matriz diagonal
        elif type_matrix == 2:
            matrix = np.triu(matrix) # Matriz triangular superior
        elif type_matrix == 3:
            matrix = np.tril(matrix) # Matriz triangular inferior
        return matrix.astype(float)
```

In [3]:
```
# EJEMPLO
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
print(f'\nMatriz completa de {A.shape}:\n {A}')
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
print(f'\nMatriz diagonal de {A.shape}:\n {A}')
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
print(f'\nMatriz triangular superior de {A.shape}:\n {A}')
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
print(f'\nMatriz triangular inferior de {A.shape}:\n {A}')
```

```
Matriz completa de (3, 3):
 [[ -8. -10. -10.]
 [  5.  -4. -10.]
 [ -4.   8.  -7.]]

Matriz diagonal de (3, 3):
 [[-2.  0.  0.]
 [ 0.  3.  0.]
 [ 0.  0.  6.]]

Matriz triangular superior de (3, 3):
 [[ 9. -6.  1.]
 [ 0. -9. -4.]
 [ 0.  0.  5.]]

Matriz triangular inferior de (3, 3):
 [[-5.  0.  0.]
 [-1. -5.  0.]
 [ 4.  5. -5.]]
```

1. Tal y como ya hemos visto en clase, la variedad de herramientas proporcionadas por el álgebra lineal son cruciales para desarrollar y fundamentar las bases de una variedad de técnicas relacionadas con el aprendizaje autom´atico. Con ella, podemos describir el proceso de propagación hacia adelante en una red neuronal, identificar mínimos locales en funciones multivariables (crucial para el proceso de retropropagación) o la descripción y empleo de métodos de reducción de la dimensionalidad, como el análisis de componentes principales (PCA), entre muchas otras aplicaciones.

Cuando trabajamos en la práctica dentro de este ámbito, la cantidad de datos que manejamos puede ser muy grande, por lo que es especialmente importante emplear algoritmos eficientes y optimizados para reducir el coste computacional en la medida de lo posible. Por todo ello, el objetivo de este ejercicio es el de ilustrar las diferentes alternativas que pueden existir para realizar un proceso relacionado con el álgebra lineal y el impacto que puede tener cada variante en términos del coste computacional del mismo. En este caso en particular, y a modo de ilustración, nos centraremos en el cálculo del determinante de una matriz.

a) [1 punto] Implementa una función, `determinante_recursivo` , que obtenga el determinante de una matriz cuadrada utilizando la definición recursiva de Laplace.

In [4]:
```python
def determinante_recursivo(matrix:np.array) -> float:
    # Comprobar si la matriz es cuadrada
    n, m = matrix.shape
    if n != m: raise ValueError("La matriz debe ser cuadrada.")
    # Caso base: si la matriz es 1x1, el determinante es el único elemento
    if n == 1:
        return matrix[0, 0]
    # Caso base: si la matriz es 2x2, aplicar la fórmula del determinante directame
    if n == 2:
```

```
        return matrix[0, 0] * matrix[1, 1] - matrix[0, 1] * matrix[1, 0]
    # Calcular el determinante recursivamente usando la expansión de Laplace
    det = 0
    for c in range(n):
        # Crear el menor eliminando la primera fila y la columna c
        minor = np.delete(np.delete(matrix, 0, axis=0), c, axis=1)
        # Aplicar la definición recursiva de Laplace
        det += ((-1) ** c) * matrix[0, c] * determinante_recursivo(minor)
    return det
```

In [5]:
```
# EJEMPLO
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
det = determinante_recursivo(matrix=A)
print(f'\nPara la matriz de {A.shape}:\n{A}\nEl determinante es: {det}.\n')
```

```
Para la matriz de (3, 3):
[[-7. -4. -6.]
 [ 4. -5. -3.]
 [ 2.  6.  3.]]
El determinante es: -153.0.
```

b) [0.5 puntos] Si A es una matriz cuadrada $n×n$ y triangular (superior o inferior, es decir, con entradas nulas por debajo o por encima de la diagonal, respectivamente), ¿existe alguna forma de calcular de forma directa y sencilla su determinante? Justifíquese la respuesta.

In [6]:
```
def determinante_matriz_triangular(matrix:np.array) -> float:
    n, m = matrix.shape
    if n != m:
        raise ValueError("La matriz debe ser cuadrada.")
    elif n < 1:
        raise ValueError("La matriz debe tener por lo menos un valor.")
    det = 1.0
    for i in range(n):
        det *= matrix[i, i]
    return det
```

> Sí, existe una forma directa y sencilla de calcular el determinante de una matriz cuadrada $n×n$ que es triangular, ya sea superior o inferior. El determinante de una matriz triangular es simplemente el producto de los elementos en su diagonal principal. **Justificación:**
>
> - Para una matriz diagonal superior:

In [7]:
```
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
det1 = determinante_recursivo(matrix=A)
det2 = determinante_matriz_triangular(matrix=A)
print(f'\nPara la matriz de {A.shape}:\n{A}\nEl determinante recursivo es: {det1}.\
```

```
Para la matriz de (3, 3):
[[ 5. -8. -9.]
 [ 0.  1. -9.]
 [ 0.  0.  9.]]
El determinante recursivo es: 45.0.
El determinante directo es: 45.0.
```

> - Para una matriz diagonal superior:

In [8]:
```
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
det1 = determinante_recursivo(matrix=A)
det2 = determinante_matriz_triangular(matrix=A)
print(f'\nPara la matriz de {A.shape}:\n{A}\nEl determinante recursivo es: {det1}.\
```

```
Para la matriz de (3, 3):
[[ 6.  0.  0.]
 [ 9. -1.  0.]
 [ 6.  6. -5.]]
El determinante recursivo es: 30.0.
El determinante directo es: 30.0.
```

c) [0.5 puntos] Determínese de forma justificada cómo alteran el determinante de una matriz $n×n$ las dos operaciones elementales siguientes:

- Intercambiar una fila (o columna) por otra fila (o columna).

> Intercambiar una fila (o columna) por otra en una matriz cuadrada $\$n×n\$$ no altera el valor absoluto del determinante de la matriz, pero puede cambiar su signo. **Justificación:**

In [9]:
```
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
det = determinante_recursivo(matrix=A)
print(f'\nPara la matriz de {A.shape}:\n{A}\nEl determinante es: {det}.\n')

# Cambiando
B = np.zeros(A.shape)
C = np.zeros(A.shape)
perm = np.array(list(itertools.permutations(list(range(A.shape[0])), A.shape[0])))
for i in perm:
    print(f'Permutacion {i}')
    for j in range(A.shape[0]):
        B[j, :] = A[i[j], :]
        C[:, j] = A[:, i[j]]
    det = determinante_recursivo(matrix=B)
    print(f'\nPara la matriz cambiada en FILAS:\n{B}\nEl determinante es: {det}.\n'
    det = determinante_recursivo(matrix=C)
    print(f'\nPara la matriz cambiada en COLUMNAS:\n{C}\nEl determinante es: {det}.
```

```
Para la matriz de (3, 3):
[[-7.  5.  4.]
 [ 6. -4. -5.]
 [ 9. -2.  9.]]
El determinante es: -77.0.

Permutacion [0 1 2]

Para la matriz cambiada en FILAS:
[[-7.  5.  4.]
 [ 6. -4. -5.]
 [ 9. -2.  9.]]
El determinante es: -77.0.


Para la matriz cambiada en COLUMNAS:
[[-7.  5.  4.]
 [ 6. -4. -5.]
 [ 9. -2.  9.]]
El determinante es: -77.0.

Permutacion [0 2 1]

Para la matriz cambiada en FILAS:
[[-7.  5.  4.]
 [ 9. -2.  9.]
 [ 6. -4. -5.]]
El determinante es: 77.0.


Para la matriz cambiada en COLUMNAS:
[[-7.  4.  5.]
 [ 6. -5. -4.]
 [ 9.  9. -2.]]
El determinante es: 77.0.

Permutacion [1 0 2]

Para la matriz cambiada en FILAS:
[[ 6. -4. -5.]
 [-7.  5.  4.]
 [ 9. -2.  9.]]
El determinante es: 77.0.


Para la matriz cambiada en COLUMNAS:
[[ 5. -7.  4.]
 [-4.  6. -5.]
 [-2.  9.  9.]]
El determinante es: 77.0.

Permutacion [1 2 0]

Para la matriz cambiada en FILAS:
[[ 6. -4. -5.]
 [ 9. -2.  9.]
```

```
 [-7.  5.  4.]]
El determinante es: -77.0.


Para la matriz cambiada en COLUMNAS:
[[ 5.  4. -7.]
 [-4. -5.  6.]
 [-2.  9.  9.]]
El determinante es: -77.0.

Permutacion [2 0 1]

Para la matriz cambiada en FILAS:
[[ 9. -2.  9.]
 [-7.  5.  4.]
 [ 6. -4. -5.]]
El determinante es: -77.0.


Para la matriz cambiada en COLUMNAS:
[[ 4. -7.  5.]
 [-5.  6. -4.]
 [ 9.  9. -2.]]
El determinante es: -77.0.

Permutacion [2 1 0]

Para la matriz cambiada en FILAS:
[[ 9. -2.  9.]
 [ 6. -4. -5.]
 [-7.  5.  4.]]
El determinante es: 77.0.


Para la matriz cambiada en COLUMNAS:
[[ 4.  5. -7.]
 [-5. -4.  6.]
 [ 9. -2.  9.]]
El determinante es: 77.0.
```

- Sumar a una fila (o columna) otra fila (o columna) multiplicada por un escalar $α$

> Sumar a una fila (o columna) otra fila (o columna) multiplicada por un escalar α no altera el determinante de una matriz cuadrada $n \times n$. **Justificación:**

In [10]:
```
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
det = determinante_recursivo(matrix=A)
print(f'\nPara la matriz de {A.shape}:\n{A}\nEl determinante es: {det}.\n')

M = np.identity(3)
pos = (0, 0)
while pos[0] == pos[1]:
```

```
    pos = (np.random.randint(0, 3), np.random.randint(0, 3))
M[pos] = np.random.randint(0, 10) # Editando la matriz, dando un valor random en cu
det = determinante_recursivo(matrix=M)
print(f'\nPara la matriz modificadora de {M.shape}:\n{M}\nEl determinante es: {det}

AM = np.dot(A, M)
det = determinante_recursivo(matrix=AM)
print(f'\nPara la matriz modificada de {AM.shape}:\n{B}\nEl determinante es: {det}.

AM = A
AM[:, pos[1]] = A[:, pos[1]] + M[pos] * A[:, pos[0]]
det = determinante_recursivo(matrix=AM)
print(f'\nPara la matriz modificada de {AM.shape}:\n{B}\nEl determinante es: {det}.
```

```
Para la matriz de (3, 3):
[[ -4.  -3. -10.]
 [  4.   9.   5.]
 [  6.   7.  -2.]]
El determinante es: 358.0.


Para la matriz modificadora de (3, 3):
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
El determinante es: 1.0.


Para la matriz modificada de (3, 3):
[[ 9. -2.  9.]
 [ 6. -4. -5.]
 [-7.  5.  4.]]
El determinante es: 358.0.


Para la matriz modificada de (3, 3):
[[ 9. -2.  9.]
 [ 6. -4. -5.]
 [-7.  5.  4.]]
El determinante es: 358.0.
```

d) [1 punto] Investiga sobre el método de eliminación de Gauss con pivoteo parcial e impleméntalo para escalonar una matriz (es decir, convertirla en una matriz triangular inferior) a partir de las operaciones elementales descritas en el apartado anterior.

> El método de eliminación de Gauss con pivoteo parcial es una técnica utilizada para resolver sistemas de ecuaciones lineales y consiste en convertir una matriz en una forma escalonada o triangular superior (o inferior) mediante una serie de operaciones elementales. El pivoteo parcial se utiliza para seleccionar el pivote (elemento principal) de cada paso de eliminación como el máximo en valor absoluto en la columna actual, lo que ayuda a reducir errores numéricos y a mejorar la estabilidad del algoritmo.

> Supongamos que tenemos la siguiente matriz original (A):
>
> $A = \begin{pmatrix}

```
2 & -1 & 1 & 2 \\
4 & 1 & -1 & 4 \\
-2 & 5 & 1 & 3 \\
6 & -3 & 4 & 10
\end{pmatrix}$ </center>
```

> Paso 1: Selección de Pivote: En la primera columna, el elemento $|4|$ es el máximo en valor absoluto, por lo que intercambiamos la primera fila con la segunda fila para llevarlo a la posición del pivote:
>
> $A = \begin{pmatrix}

```
4 & 1 & -1 & 4 \\
2 & -1 & 1 & 2 \\
-2 & 5 & 1 & 3 \\
6 & -3 & 4 & 10
\end{pmatrix}$ </center>
```

> Paso 2: Eliminación hacia adelante: Utilizamos el primer elemento de la primera fila como pivote para eliminar los elementos debajo de él:
>
> $A = \begin{pmatrix}

```
4 & 1 & -1 & 4 \\
0 & -1.5 & 1.5 & 0 \\
0 & 6.5 & 0.5 & 5 \\
0 & -7 & 8 & -6
\end{pmatrix}$ </center>
```

> Paso 3: Nuevo pivote y eliminación: En la segunda columna, el elemento $|-7|$ es el máximo en valor absoluto, por lo que intercambiamos la tercera fila con la cuarta fila:
>
> $A = \begin{pmatrix}

```
4 & 1 & -1 & 4 \\
0 & -1.5 & 1.5 & 0 \\
0 & -7 & 8 & -6 \\
0 & 6.5 & 0.5 & 5
\end{pmatrix}$ </center>
```

> Paso 4: Eliminación hacia adelante nuevamente: Utilizamos el segundo elemento de la segunda fila como pivote para eliminar los elementos debajo

> de él:
>
> $A = \begin{pmatrix}
>
> 4 & 1 & -1 & 4 \\
> 0 & -7 & 8 & -6 \\
> 0 & 0 & 9.571 & 4.571 \\
> 0 & 0 & 15 & 9
>
> \end{pmatrix}$

> Finalmente, obtenemos la matriz escalonada (triangular inferior) A luego de aplicar el método de eliminación de Gauss con pivoteo parcial.

e) [0.5 puntos] ¿Cómo se podría calcular el determinante de una matriz haciendo beneficio de la estrategia anterior y del efecto de aplicar las operaciones elementales pertinentes? Implementa una nueva función, determinante gauss, que calcule el determinante de una matriz utilizando eliminación gaussiana.

> Aplicando el método de eliminación de Gauss, se pasa de una matriz cuadrada completa a una matriz cuadrada triangular superior o inferior, anteriormente dicho que la determinante de un matriz triangular es igual al multiplo de los valores en la diagonal.

In [11]:
```python
def metodo_eliminacion_Gauss(matrix:np.array, type:str='up') -> np.array:
    n, m = matrix.shape
    if n != m:
        raise ValueError("La matriz debe ser cuadrada.")
    elif n < 1:
        raise ValueError("La matriz debe tener por lo menos un valor.")
    matrix_tr = np.copy(matrix)
    if type == 'up':
        for i in range(n):
            if matrix_tr[i,i] == 0: raise ValueError(f'Valor {i}x{i} es 0.')
            else:
                for j in range(i+1, n):
                    factor = matrix_tr[j, i]/matrix_tr[i, i]
                    matrix_tr[j, :] = matrix_tr[j, :] - factor*matrix_tr[i, :]
    elif type == 'down':
        for i in range(n):
            if matrix_tr[i,i] == 0: raise ValueError(f'Valor {i}x{i} es 0.')
            else:
                for j in range(i+1, n):
                    factor = matrix_tr[i, j]/matrix_tr[i, i]
                    matrix_tr[:, j] = matrix_tr[:, j] - factor*matrix_tr[:, i]
    return matrix_tr
```

In [12]:
```python
A = matriz_entera_cuadrada_random(min_value=-10, max_value=10, size_matrix=3, type_
det = determinante_recursivo(matrix=A)
```

```
print(f'\nPara la matriz de {A.shape}:\n{A}\nEl determinante es: {det}.\n')

B = metodo_eliminacion_Gauss(matrix=A, type='up')
det = determinante_matriz_triangular(matrix=B)
print(f'\nPara la matriz triangular de {B.shape}:\n{B}\nEl determinante es: {det}.\
```

```
Para la matriz de (3, 3):
[[ 5. -3. -6.]
 [-2.  0. -7.]
 [ 7.  7. -9.]]
El determinante es: 530.0.


Para la matriz triangular de (3, 3):
[[ 5.00000000e+00 -3.00000000e+00 -6.00000000e+00]
 [ 0.00000000e+00 -1.20000000e+00 -9.40000000e+00]
 [ 0.00000000e+00 -1.77635684e-15 -8.83333333e+01]]
El determinante es: 530.0.
```

f) [0.5 puntos] Obtén la complejidad computacional asociada al cálculo del determinante con la definición recursiva y con el método de eliminación de Gauss con pivoteo parcial.

> - Complejidad de la función `determinante_recursivo` es:
>
> La recursividad está dada por: $$T(n)=n*T(n-1)$$ Dando como resultado parcial desde $i=n$ hasta $i=2$: $$T(n)=n!*T(2)$$ Sabiendo que al llegar a $i=2$ se resuelve con operaciones básicas que es de complejidad $O(1)$. Entonces, da como resultado total una complejidada de $O(n!)$

> - Complejidad de la función `metodo_eliminacion_Gauss` y `determinante_matriz_triangular` es:
>
> En el Método de eliminación de Gauss, se pueden observar dos blucles, uno de n ejecuciones como máximo y el otro como $n-1$ ejecuciones como máximo. Dando que: $$T(n)=n(n-1)$$ En el Determinante de una matriz ztriangular, solo se usan operaciones básicas, siendo de complejidad $O(1)$ Entonces, da como resultado total una complejidada de $O(n^2)$

g) [1 punto] Utilizando `numpy.random.rand`, genera matrices cuadradas aleatorias de la forma $A_n ∈ \mathbb{R}^{n×n}$, para $2 ≤ n ≤ 10$, y confecciona una tabla comparativa del tiempo de ejecución asociado a cada una de las variantes siguientes, interpretando los resultados:

> Ya que el método `numpy.random.rand` solo da valores incluidos en $[0, 1)$, he preferido utilizar el `numpy.random.randint` que permite modificar el

> mínimo y máximo, dando valores negativos y enteros facilitando la comparación mecánica.

In [13]:
```
n = 6 # Tamaño de la matriz
k = 4 # Número de matrices

value = True # Verificador
A = [] # Conjunto de matrices

while value:
    for i in range(k):
        A.append(matriz_entera_cuadrada_random(size_matrix=n)) # Revisar que la dia
        for j in range(n):
            if A[i][j, j] == 0:
                value = True
                pass
            else: value = False
tiempos = []
```

- Utilizando determinante recursivo.

In [14]:
```
tiempos_ = []
for a in A:
    start_time = time.time_ns()

    det = determinante_recursivo(matrix=a)
    print(f'\nPara la matriz de {a.shape}:\n{a}\nEl determinante es: {det}.\n')

    pause_time = time.time_ns()
    print(f'\nTiempo transcurrido: {pause_time - start_time} ns.')
    tiempos_.append(pause_time - start_time)
tiempos.append(tiempos_)
```

```
Para la matriz de (6, 6):
[[ -2.   7.   7. -10. -10.  -7.]
 [  9.   9.  -5.   4.   5.  -1.]
 [  8.  -6.  -8.  -5.  -6.  -1.]
 [  7.   7.  -7. -10.  -8.   9.]
 [  9.  -8.  -5.   3.   4.  -8.]
 [  6.  -3.  -8.   8.  -1.  -9.]]
El determinante es: 2091.0.


Tiempo transcurrido: 12009800 ns.

Para la matriz de (6, 6):
[[  8.   2.  -3.   9.  -9.  -4.]
 [ -7. -10.  -7.  -9.   7.  -6.]
 [  9.   5.   0.   6.  -6.  -2.]
 [ -8.   3.  -7.   2.  -3.  -8.]
 [  7.   7.  -5.   5.  -2.   2.]
 [  8.  -2.  -7.   0.  -3.   5.]]
El determinante es: 330395.0.


Tiempo transcurrido: 4179300 ns.

Para la matriz de (6, 6):
[[  2.  -7.   3.   8.  -3.   6.]
 [  5.  -4.   7.   6.  -7.   1.]
 [  9.  -2.   2.  -6.   1.   6.]
 [ -2.   0.  -9. -10.   5. -10.]
 [ -9.  -6.   4.  -5.   8.   9.]
 [  9.  -3.   6.  -4.  -4.   0.]]
El determinante es: 5763.0.


Tiempo transcurrido: 7690300 ns.

Para la matriz de (6, 6):
[[  7.   5.  -5.   2.  -1.  -2.]
 [  7.   3.  -4.  -6.   8.   9.]
 [ -9.   8.   2.   6.  -1.   3.]
 [  8.   1.   3.   3.  -8.  -6.]
 [-10.   6.  -3.   6.   7.  -8.]
 [ -2.   7. -10.   6.   6.  -1.]]
El determinante es: -322877.0.


Tiempo transcurrido: 5998100 ns.
```

- Empleando determinante gauss.

In [15]:
```
tiempos_ = []
for a in A:
    start_time = time.time_ns()

    det = determinante_matriz_triangular(metodo_eliminacion_Gauss(matrix=a))
```

```
    print(f'\nPara la matriz de {a.shape}:\n{a}\nEl determinante es: {det}.\n')

    pause_time = time.time_ns()
    print(f'\nTiempo transcurrido: {pause_time - start_time} ns.')
    tiempos_.append(pause_time - start_time)
tiempos.append(tiempos_)
```

```
Para la matriz de (6, 6):
[[ -2.   7.   7. -10. -10.  -7.]
 [  9.   9.  -5.   4.   5.  -1.]
 [  8.  -6.  -8.  -5.  -6.  -1.]
 [  7.   7.  -7. -10.  -8.   9.]
 [  9.  -8.  -5.   3.   4.  -8.]
 [  6.  -3.  -8.   8.  -1.  -9.]]
El determinante es: 2090.999999999244.


Tiempo transcurrido: 903200 ns.

Para la matriz de (6, 6):
[[  8.   2.  -3.   9.  -9.  -4.]
 [ -7. -10.  -7.  -9.   7.  -6.]
 [  9.   5.   0.   6.  -6.  -2.]
 [ -8.   3.  -7.   2.  -3.  -8.]
 [  7.   7.  -5.   5.  -2.   2.]
 [  8.  -2.  -7.   0.  -3.   5.]]
El determinante es: 330395.0000000011.


Tiempo transcurrido: 1091300 ns.

Para la matriz de (6, 6):
[[  2.  -7.   3.   8.  -3.   6.]
 [  5.  -4.   7.   6.  -7.   1.]
 [  9.  -2.   2.  -6.   1.   6.]
 [ -2.   0.  -9. -10.   5. -10.]
 [ -9.  -6.   4.  -5.   8.   9.]
 [  9.  -3.   6.  -4.  -4.   0.]]
El determinante es: 5762.999999999911.


Tiempo transcurrido: 0 ns.

Para la matriz de (6, 6):
[[  7.   5.  -5.   2.  -1.  -2.]
 [  7.   3.  -4.  -6.   8.   9.]
 [ -9.   8.   2.   6.  -1.   3.]
 [  8.   1.   3.   3.  -8.  -6.]
 [-10.   6.  -3.   6.   7.  -8.]
 [ -2.   7. -10.   6.   6.  -1.]]
El determinante es: -322877.0000000002.


Tiempo transcurrido: 908500 ns.
```

- Haciendo uso de la función preprogramada `numpy.linalg.det`.

In [16]:
```
tiempos_ = []
for a in A:
    start_time = time.time_ns()

    det = np.linalg.det(a)
    print(f'\nPara la matriz de {a.shape}:\n{a}\nEl determinante es: {det}.\n')

    pause_time = time.time_ns()
    print(f'\nTiempo transcurrido: {pause_time - start_time} ns.')
    tiempos_.append(pause_time - start_time)
tiempos.append(tiempos_)
```

```
Para la matriz de (6, 6):
[[ -2.   7.   7. -10. -10.  -7.]
 [  9.   9.  -5.   4.   5.  -1.]
 [  8.  -6.  -8.  -5.  -6.  -1.]
 [  7.   7.  -7. -10.  -8.   9.]
 [  9.  -8.  -5.   3.   4.  -8.]
 [  6.  -3.  -8.   8.  -1.  -9.]]
El determinante es: 2091.00000000012.


Tiempo transcurrido: 1002800 ns.

Para la matriz de (6, 6):
[[  8.   2.  -3.   9.  -9.  -4.]
 [ -7. -10.  -7.  -9.   7.  -6.]
 [  9.   5.   0.   6.  -6.  -2.]
 [ -8.   3.  -7.   2.  -3.  -8.]
 [  7.   7.  -5.   5.  -2.   2.]
 [  8.  -2.  -7.   0.  -3.   5.]]
El determinante es: 330395.00000000035.


Tiempo transcurrido: 997300 ns.

Para la matriz de (6, 6):
[[  2.  -7.   3.   8.  -3.   6.]
 [  5.  -4.   7.   6.  -7.   1.]
 [  9.  -2.   2.  -6.   1.   6.]
 [ -2.   0.  -9. -10.   5. -10.]
 [ -9.  -6.   4.  -5.   8.   9.]
 [  9.  -3.   6.  -4.  -4.   0.]]
El determinante es: 5763.000000000136.


Tiempo transcurrido: 0 ns.

Para la matriz de (6, 6):
[[  7.   5.  -5.   2.  -1.  -2.]
 [  7.   3.  -4.  -6.   8.   9.]
 [ -9.   8.   2.   6.  -1.   3.]
 [  8.   1.   3.   3.  -8.  -6.]
 [-10.   6.  -3.   6.   7.  -8.]
 [ -2.   7. -10.   6.   6.  -1.]]
El determinante es: -322877.0000000003.


Tiempo transcurrido: 0 ns.
```

> La tabla de tiempo de ejecución es:

In [17]:
```python
tiempos_df = pd.DataFrame({
    'Recursivo': tiempos[0],
    'Gauss': tiempos[1],
    'Numpy': tiempos[2]
```

```
})
print("Tiempo de ejecución.")
display(tiempos_df)
```

```
Tiempo de ejecución.
```

	Recursivo	Gauss	Numpy
0	12009800	903200	1002800
1	4179300	1091300	997300
2	7690300	0	0
3	5998100	908500	0

2. En este ejercicio trabajaremos con el método de descenso de gradiente, el cual constituye otra herramienta crucial, en esta ocasión de la rama del cálculo, para el proceso de retropropagación asociado al entrenamiento de una red neuronal.

a) [1 punto] Prográmese en Python el método de descenso de gradiente para funciones de n variables. La función deberá tener como parámetros de entradas:

- El gradiente de la función que se desea minimizar ∇f (puede venir dada como otra función previamente implementada, `grad_f`, con entrada un vector, representando el punto donde se quiere calcular el gradiente, y salida otro vector, representando el gradiente de f en dicho punto).
- Un valor inicial $x_0 \in \mathbb{R}^n$ (almacenado en un vector de n componentes).
- El ratio de aprendizaje γ (que se asume constante para cada iteración).
- Un parámetro de tolerancia tol (con el que finalizar el proceso cuando $\|\nabla f(x)\|_2 < tol$).
- Un número máximo de iteraciones `maxit` (con el fin de evitar ejecuciones indefinidas en caso de divergencia o convergencia muy lenta).

La salida de la función deberá ser la aproximación del x que cumple $f'(x) \approx 0$, correspondiente a la última iteración realizada en el método.

In [18]:
```
def gradient_descent(grad_f:callable, x0:np.ndarray, gamma:float, tol:float, maxit:
    dic = {
        'Iteración': [],
        'Xi': [],
        'Tolerancia': []
    }
    x = np.array(x0, dtype=float)
    for i in range(int(maxit)):
        grad = grad_f(*x)
        dic['Iteración'].append(i+1)
        dic['Xi'].append(x)
        dic['Tolerancia'].append(np.linalg.norm(grad))
        if np.linalg.norm(grad) < tol:
            break
```

```
        x = x - gamma*grad.flatten()
    return x, i+1, dic
```

b) Sea la función $f : \mathbb{R} → \mathbb{R}$ dada por: $$f(x) = 3x^4 + 4x^3 − 12x^2 + 7$$

In [19]:
```
x = sp.symbols('x') # Variables independientes

f_sym = 3*x**4 + 4*x**3 - 12*x**2 + 7 # Funcion
grad_f_sym = sp.diff(f_sym, x) # Gradiente

f = sp.lambdify(x, f_sym, 'numpy') # Funcion lambda de f
grad_f = sp.lambdify(x, grad_f_sym, 'numpy') # Funcion lambda del gradiente de f
```

- [0.5 puntos] Aplica el método sobre $f(x)$ con $x_0 = 3$ $γ = 0.001$, $tol=1e^{-12}$, $maxit=1e^5$.

In [20]:
```
x0 = [3]
gamma = 0.001
tol = 1e-12
maxit = 1e5

result, i, dic = gradient_descent(grad_f=grad_f, x0=x0, gamma=gamma, tol=tol, maxit
print(f'\nPara f(x):\nPunto inicial: {x0}.\nUn gamma: {gamma}.\nUna tolerancia máxi
df = pd.DataFrame(dic)
df.set_index('Iteración', inplace=True)
display(df)
```

```
Para f(x):
Punto inicial: [3].
Un gamma: 0.001.
Una tolerancia máxima: 1e-12.

Se obtuvo:
En una iteración de 832/100000.0.
Un resultado de [1.] aproximado.
```

Iteración	Xi	Tolerancia
1	[3.0]	3.600000e+02
2	[2.64]	2.410721e+02
3	[2.3989278720000002]	1.771498e+02
4	[2.2217780988709572]	1.375212e+02
5	[2.084256894881889]	1.107587e+02
...	...	...
828	[1.000000000000032]	1.151079e-12
829	[1.0000000000000309]	1.115552e-12
830	[1.0000000000000298]	1.069367e-12
831	[1.0000000000000286]	1.030287e-12
832	[1.0000000000000275]	9.912071e-13

832 rows × 2 columns

- [0.5 puntos] Aplica de nuevo el método sobre $f(x)$ con $x_0 = 3$, $γ = 0.01$, $tol=1e^{-12}$, $maxit=1e^5$.

In [21]:
```
x0 = [3]
gamma = 0.01
tol = 1e-12
maxit = 1e5

result, i, dic = gradient_descent(grad_f=grad_f, x0=x0, gamma=gamma, tol=tol, maxit
print(f'\nPara f(x):\nPunto inicial: {x0}.\nUn gamma: {gamma}.\nUna tolerancia máxi
df = pd.DataFrame(dic)
df.set_index('Iteración', inplace=True)
display(df)
```

```
Para f(x):
Punto inicial: [3].
Un gamma: 0.01.
Una tolerancia máxima: 1e-12.

Se obtuvo:
En una iteración de 32/100000.0.
Un resultado de [-2.] aproximado.
```

Iteración	Xi	Tolerancia
1	[3.0]	3.600000e+02
2	[-0.6000000000000001]	1.612800e+01
3	[-0.7612800000000001]	1.993091e+01
4	[-0.9605891420101018]	2.349053e+01
5	[-1.1954944129044498]	2.533904e+01
6	[-1.4488848365458096]	2.346529e+01
7	[-1.683537687896648]	1.715670e+01
8	[-1.8551046987375075]	9.209287e+00
9	[-1.9471975697570583]	3.636256e+00
10	[-1.9835601277416384]	1.167508e+00
11	[-1.995235207311065]	3.417042e-01
12	[-1.9986522490786303]	9.692911e-02
13	[-1.9996215401762607]	2.724051e-02
14	[-1.999893945316755]	7.635262e-03
15	[-1.9999702979402771]	2.138495e-03
16	[-1.9999916828939535]	5.988275e-04
17	[-1.9999976711688026]	1.676755e-04
18	[-1.9999993479240106]	4.694945e-05
19	[-1.9999998174184679]	1.314587e-05
20	[-1.999999948877151]	3.680845e-06
21	[-1.9999999856856008]	1.030637e-06
22	[-1.999999995991968]	2.885783e-07
23	[-1.9999999988777513]	8.080192e-08
24	[-1.9999999996857705]	2.262453e-08
25	[-1.9999999999120157]	6.334879e-09
26	[-1.9999999999753646]	1.773749e-09
27	[-1.9999999999931022]	4.966338e-10
28	[-1.9999999999980684]	1.390816e-10
29	[-1.9999999999994593]	3.893774e-11

	Xi	Tolerancia
Iteración		
30	[-1.9999999999998488]	1.087841e-11
31	[-1.9999999999999576]	3.062439e-12
32	[-1.9999999999999882]	8.384404e-13

- [0.5 puntos] Contrasta e interpreta los dos resultados obtenidos en los apartados anteriores y compáralos con los mínimos locales obtenidos analíticamente. ¿Qué influencia puede llegar a tener la elección del ratio de aprendizaje $γ$?

> En ambos casos, el punto inicial como la tolerancia máxima y el máximo número de iteraciones es el mismo. Dejando evidente que la diferencia de resultados dependió en este caso de γ.
>
> - Para un valor de γ demasiado pequeño, puede llevar a una convergencia demasiada lenta como se observa en el primer caso donde se necesitó 832 iteraciones para alcanzar un resultado.
> - Para un valor de γ demasiado grande, puede provocar que el algoritmo diverja o que oscile alrededor del mínimo como se observa en el segundo caso donde se necesitó solo 32 iteraciones para alcanzar un resultado pero es mayor que el primero.

- [0.5 puntos] Aplica nuevamente el método sobre $f(x)$ con $x_0 = 3$, $γ = 0.1$, $tol=1e^{-12}$, $maxit=1e^5$. Interpreta el resultado.

In [22]:
```
x0 = [3]
gamma = 0.1
tol = 1e-12
maxit = 1e5

result, i, dic = gradient_descent(grad_f=grad_f, x0=x0, gamma=gamma, tol=tol, maxit
print(f'\nPara f(x):\nPunto inicial: {x0}.\nUn gamma: {gamma}.\nUna tolerancia máxi
df = pd.DataFrame(dic)
df.set_index('Iteración', inplace=True)
display(df)
```

```
<lambdifygenerated-2>:2: RuntimeWarning: overflow encountered in scalar power
  return 12*x**3 + 12*x**2 - 24*x
<lambdifygenerated-2>:2: RuntimeWarning: invalid value encountered in scalar subtrac
t
  return 12*x**3 + 12*x**2 - 24*x
```

```
Para f(x):
Punto inicial: [3].
Un gamma: 0.1.
Una tolerancia máxima: 1e-12.

Se obtuvo:
En una iteración de 100000/100000.0.
Un resultado de [nan] aproximado.
```

Iteración	Xi	Tolerancia
1	[3.0]	3.600000e+02
2	[-33.0]	4.173840e+05
3	[41705.4]	8.704995e+14
4	[-87049951065956.78]	7.915655e+42
5	[7.915654682433035e+41]	5.951710e+126
...	...	...
99996	[nan]	NaN
99997	[nan]	NaN
99998	[nan]	NaN
99999	[nan]	NaN
100000	[nan]	NaN

100000 rows × 2 columns

> En este caso el valor de γ es mayor que el segundo caso, aplicando que oscile alrededor del mínimo pero también se aleja al mismo. Da pasos demasiado grandes en cada iteración, lo que puede llevar a una divergencia o a resultados numéricamente inestables.

- [0.5 puntos] Finalmente, aplica el método sobre $f(x)$ con $x_0 = 0$, $γ = 0.001$, $tol=1e^{-12}$, $maxit=1e^5$. Interpreta el resultado y compáralo con el estudio analítico de f. ¿Se trata de un resultado deseable? ¿Por qué? ¿A qué se debe este fenómeno?

In [23]:
```
x0 = [0]
gamma = 0.001
tol = 1e-12
maxit = 1e5

result, i, dic = gradient_descent(grad_f=grad_f, x0=x0, gamma=gamma, tol=tol, maxit
```

```
print(f'\nPara f(x):\nPunto inicial: {x0}.\nUn gamma: {gamma}.\nUna tolerancia máxi
df = pd.DataFrame(dic)
df.set_index('Iteración', inplace=True)
display(df)
```

```
Para f(x):
Punto inicial: [0].
Un gamma: 0.001.
Una tolerancia máxima: 1e-12.

Se obtuvo:
En una iteración de 1/100000.0.
Un resultado de [0.] aproximado.
```

Iteración	Xi	Tolerancia
1	[0.0]	0.0

> En este caso tenemos el mismo valor de γ que el primer caso, lo que cambia es el punto inicial. Se obseva que se obtuvo una sola iteración siendo la tolerancia menor a la máxima permitida, que la tolerancia es 0 dando a conocer que el punto inicial $x_0=0$ es el mínimo que buscabamos.

c) Sea la función $g : \mathbb{R}^2 → \mathbb{R}$ dada por: $$g(x, y) = x^2 + y^3 + 3xy + 1$$

In [24]:
```
x, y = sp.symbols('x y') # Variables independientes

g_sym = x**2 + y**3 + 3*x*y + 1 # Funcion
grad_g_sym = sp.Matrix([sp.diff(g_sym, var) for var in (x, y)]) # Gradiente

g = sp.lambdify((x, y), g_sym, 'numpy') # Funcion lambda de g
grad_g = sp.lambdify((x, y), grad_g_sym, 'numpy') # Funcion lambda del gradiente de
```

- [0.5 puntos] Aplíquese el método sobre $g(x, y)$ con $x_0 = (−1, 1)$, $γ = 0.01$, $tol=1e^{-12}$, $maxit=1e^5$.

In [25]:
```
x0 = [-1, 1]
gamma = 0.01
tol = 1e-12
maxit = 1e5

result, i, dic = gradient_descent(grad_f=grad_g, x0=x0, gamma=gamma, tol=tol, maxit
print(f'\nPara g(x):\nPunto inicial: {x0}.\nUn gamma: {gamma}.\nUna tolerancia máxi
df = pd.DataFrame(dic)
df.set_index('Iteración', inplace=True)
display(df)
```

```
Para g(x):
Punto inicial: [-1, 1].
Un gamma: 0.01.
Una tolerancia máxima: 1e-12.

Se obtuvo:
En una iteración de 3140/100000.0.
Un resultado de [-2.25  1.5 ] aproximado.
```

Iteración	Xi	Tolerancia
1	[-1.0, 1.0]	1.000000e+00
2	[-1.01, 1.0]	9.804591e-01
3	[-1.0198, 1.0003]	9.630241e-01
4	[-1.0294130000000001, 1.0008759972999999]	9.474429e-01
5	[-1.0388510199190002, 1.0017058044408618]	9.334886e-01
...	...	...
3136	[-2.2499999999989093, 1.4999999999995965]	1.034972e-12
3137	[-2.249999999998919, 1.4999999999996]	1.025717e-12
3138	[-2.249999999998929, 1.4999999999996037]	1.016463e-12
3139	[-2.249999999998938, 1.4999999999996072]	1.007733e-12
3140	[-2.2499999999989475, 1.4999999999996108]	9.990033e-13

3140 rows × 2 columns

- [0.5 puntos] ¿Qué ocurre si ahora partimos de $x_0 = (0, 0)$? ¿Se obtiene un resultado deseable?

In [26]:
```
x0 = [0, 0]
gamma = 0.01
tol = 1e-12
maxit = 1e5

result, i, dic = gradient_descent(grad_f=grad_g, x0=x0, gamma=gamma, tol=tol, maxit
print(f'\nPara g(x):\nPunto inicial: {x0}.\nUn gamma: {gamma}.\nUna tolerancia máxi
df = pd.DataFrame(dic)
df.set_index('Iteración', inplace=True)
display(df)
```

```
Para g(x):
Punto inicial: [0, 0].
Un gamma: 0.01.
Una tolerancia máxima: 1e-12.

Se obtuvo:
En una iteración de 1/100000.0.
Un resultado de [0. 0.] aproximado.
```

Iteración	Xi	Tolerancia
1	[0.0, 0.0]	0.0

> Los valores de punto inicial ya eran los adecuados y deseados.

- [0.5 puntos] Realícese el estudio analítico de la función y utilícese para explicar y contrastar los resultados obtenidos en los dos apartados anteriores.

> 1. Dominio: La función está definida para cualquier valore de x e y en los números reales, es decir, el dominio es $\mathbb{R}^2$.
> 2. Simetría: La función no es simétrica respecto al origen, ya que $g(x,y) \neq g(-x,-y)$.
> 3. Puntos críticos: Los puntos críticos se encuentran donde el gradiente de la función es cero. Calculamos el gradiente de g e igualamos a 0 dando que $3y^2+3x=0$ lo que indica que no tiene una sola solución.
>
> - Primer apartado.
>
> Muestra un mínimo "global" en el punto $[-2.25, 1.5]$ con un valor de función de 0. El algoritmo de descenso de gradiente logró encontrar este mínimo en 3140 iteraciones con un valor de γ de 0.01 y una tolerancia máxima de 1×10^{-12}.
>
> - Segundo apartado.
>
> Muestra un mínimo "global" en el punto $[0, 0]$ con un valor de función de 1. El algoritmo de descenso de gradiente logró encontrar este mínimo en 1 iteración con un valor de γ de 0.01 y una tolerancia máxima de 1×10^{-12}.

FIN