Ejercicios_entragables_michael_laudrup_luis_gonzalez

July 2, 2025

Nombre alumno: Michael Laudrup Luis González

Repositorio GIT asociado: https://github.com/MichaeLaudrup/Maths_for_IA_VIU/tree/main/T

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

[4]:
```python
import  numpy as np

def determinante_recursivo(matriz):
    matriz = np.array(matriz, dtype=float)

    if matriz.shape[0] == 2 and matriz.shape[1] == 2:
        return (matriz[0, 0] * matriz[1, 1]) - (matriz[0, 1] * matriz[1, 0])
    else:
        row_idx = 0
        det = 0.0
        for col_idx, ele in enumerate(matriz[row_idx]):
            minor = np.delete(np.delete(matriz, 0, axis=0), col_idx, axis=1)
            signo = (-1)**(col_idx + row_idx)
            det += ele * signo * determinante_recursivo(minor)
        return det
```

1.b [0.5 puntos] Si A es una matriz cuadrada $n \times n$ 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.

En el caso de nuestro algoritmo anteriormente implementado tenemos que fijarnos en el caso de que la matriz objeto de estudio sea mayor de 2x2 se ejecutará el siguiente código:

```
row_idx = 0
det = 0
for col_idx, ele in enumerate(matriz[row_idx]):
    minor = np.delete(np.delete(matriz, 0, axis=0), col_idx, axis=1)
    det += ele * (-1)**((col_idx+row_idx)) * determinante_recursivo(minor)
return det
```

Al ser una matriz cuadrada triangular superior o inferior, ya no es necesario el bucle “for” sino que bastaría con siempre coger el elemento de la fila que más cero tiene, es decir:

- Para una matriz cuadrada triangular superior: Siempre cogeremos el último elemento de la última fila, quedandonos:

```
row_idx = matriz.shape[0] - 1 # número de filas (cogemos ultima posición)
col_idx = matriz.shape[1] - 1 # número de columnas (cogemos última posición)
minor = np.delete(np.delete(matriz, row_idx, axis=0), col_idx, axis=1)
det = ele * (-1)**((col_idx+row_idx)) * determinante_recursivo(minor)
```

Lo que apreciaremos es que a medida que vamos bajando en la recursividad, se sucede lo mismo en cadena, por lo que podemos concluir que realmente la operación quedará como la multiplicación de los elementos de la diagonal principal.

- Para una matriz cuadrada triangular inferior: Sucederá exactamente lo mismo que el anterior caso, pero en lugar de coger la última posición de la última fila en cada paso recursivo, lo que haremos será coger el primer elemento de la primera fila, por lo que:

```
row_idx = 0
col_idx = 0
```

En conclusión, para matrices triangulares superiores o inferiores, el determinante se puede calcular directamente como el producto de los elementos de la diagonal principal, ya que todos los menores serán matrices de orden inferior que también son triangulares y su determinante será cero, excepto el último paso recursivo.

Nota importante: Se ha intentado dar una justificación más asociada a la implementación especifica de este algoritmo para aportar un razonamiento diferente al que se suele dar en los libros de texto, pero la justificación general es la misma, ya que al final el determinante de una matriz triangular superior o inferior es el producto de los elementos de la diagonal principal.

1.c [0.5 puntos] Determínese de forma justificada cómo alteran el determinante de una matriz $n \times n$ las dos operaciones elementales siguientes:

- Intercambiar una fila (o columna) por otra fila (o columna): En este caso cada vez que intercambiamos filas o columnas invertimos el signo del determinante. Un ejemplo sencillo se puede ver con una matriz 2x2, si tenemos la matriz:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \rightarrow \quad \det(A) = ad - bc$$

Luego, si intercambiamos las filas, obtenemos:

$$A' = \begin{pmatrix} c & d \\ a & b \end{pmatrix} \quad \rightarrow \quad \det(A') = cb - da = -(ad - bc) = -\det(A)$$

Para el caso de matrices mayores, el efecto es el mismo.

- Suma a una fila (o columna) otra fila (o columna) multiplicada por un escalar α : Al final estas operaciones lo que hacen es darnos una combinación lineal del vector fila o vector columna sobre el que aplicamos la operación, es decir, que realmente estaríamos obteniendo un vector equivalente por lo que no estaríamos perdiendo ni ganando información, solo transformandola, entonces matemáticamente hablando siguen siendo los mismos vectores, esto se extrapolará en el calculo del determinante, el cual, no se verá alterado por esta operación.

1.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.

[5]:
```
import numpy as np

matriz_B = np.array([[4, 2, 3],
            [4, 13, 6],
            [7, 8, 9]])

import numpy as np

matriz_B = np.array([[4, 2, 3],
                     [4, 13, 6],
                     [7, 8, 9]])

import numpy as np

def gauss_elimination(matrix):
    """
    Triangulariza inferiormente la matriz y devuelve el número de intercambios␣
 ↪realizados.
    """
    A = np.array(matrix, dtype=float)
    n = A.shape[0]

    if n <= 1:
        return A, 0

    swaps = 0
    pivot_col = n - 1
    max_row = np.argmax(np.abs(A[:, pivot_col]))

    if max_row != n - 1:
        A[[n - 1, max_row]] = A[[max_row, n - 1]]
        swaps = 1

    if abs(A[n - 1, n - 1]) > 1e-12:
        for row_idx in range(n - 1):
            factor = A[row_idx, n - 1] / A[n - 1, n - 1]
            A[row_idx] = A[row_idx] - (A[n - 1] * factor)

    sub_matrix = A[:-1, :-1]
    triangulated_sub_matrix, lower_level_swaps = gauss_elimination(sub_matrix)

    A[:-1, :-1] = triangulated_sub_matrix
    total_swaps = swaps + lower_level_swaps
```

```
    return A, total_swaps

gauss_elimination(matriz_B)
```

[5]:
```
(array([[ 1.60869565,  0.        ,  0.        ],
        [-0.66666667,  7.66666667,  0.        ],
        [ 7.        ,  8.        ,  9.        ]]),
 0)
```

1.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.

[6]:
```
def determinante_gauss(matriz):
    matriz = np.array(matriz, dtype=float)
    if(matriz.shape[0] == 2 and matriz.shape[1] == 2):
        return (matriz[0][0] * matriz[1][1]) - (matriz[0][1] * matriz[1][0])

    matriz, total_swaps = gauss_elimination(matriz)
    det = 1
    for i in range(matriz.shape[0]):
        det *= matriz[i][i]

    return det * (-1)**total_swaps
```

1.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.

Coste computacional determinante recursivo

En el caso de la función `determinante_recursivo`, tenemos que:

- El caso base tiene coste $O(1)$, ya que simplemente devuelve un valor.
- Para una matriz $n \times n$, el bucle `for` itera n veces, y en cada iteración se llama recursivamente a `determinante_recursivo` con una matriz de tamaño $(n-1) \times (n-1)$. Por lo tanto, la relación de recurrencia es:

$$T(n) = n \cdot T(n-1) + operaciones_adicionales$$

Entiendosé como operaciones, por ejemplo, `np.delete` que tiene un coste de $O(n^2)$, sin embargo, el término dominante proviene de las llamadas recursivas. En esta recursividad tenemos que:

- Para $T(n)$, hacemos n llamadas a $T(n-1)$ -> $T(n) = n \cdot T(n-1)$
- Para $T(n-1)$, hacemos $(n-1)$ llamadas a $T(n-2)$ -> $T(n-1) = (n-1) \cdot T(n-2)$
- Y así sucesivamente hasta llegar al caso base $T(2)$. -> $T(n) = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot T(2)$

De todo esto podemos concluir que el coste total de la función `determinante_recursivo` es $O(n!)$, ya que el número de llamadas recursivas crece factorialmente con el tamaño de la matriz.

Coste computacional determinante gaussiano

La complejidad del algoritmo determinante_gauss está casi enteramente determinada por el coste de la función gauss_elimination. Una vez que la matriz está en su forma triangular superior, calcular el determinante es muy rápido (un solo bucle de coste O(n))

Analicemos la complejidad de `gauss_elimination`: - La función se llama a sí misma recursivamente con una matriz cada vez más pequeña, va primero con n, luego con $n-1$, y así sucesivamente - En cada llamada hay un bucle for especializado en poner a cero los elementos de la primera columna, ejecutandose $k-1$ veces - Dentro del bucle la línea `A[row_idx] = A[row_idx] - (A[0] * factor)` realiza una operación vectorial con coste $O(k)$. - Por lo tanto, la primera parte de la función `gauss_elimination` tiene un coste de $O(k^2)$, ya que se ejecuta $k-1$ veces y cada vez realiza una operación de coste $O(k)$. - Tras esto se realiza una llamada recursiva a `gauss_elimination` con una matriz de tamaño $k-1$, lo que nos da la relación de recurrencia:

$$T(k) = O(k^2) + T(k-1)$$

- Al expandir esta relación, obtenemos:

$$T(k) = O(k^2) + O((k-1)^2) + O((k-2)^2) + ... + O(1^2)$$

La suma de los cuadrados de los primeros k números es conocida y se puede expresar como:

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

Por lo tanto, la complejidad total de `gauss_elimination` es:

$$T(k) = O(k^3)$$

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

- Utilizando `determinante_recursivo`.
- Empleando `determinante_gauss`.
- Haciendo uso de la función preprogramada `numpy.linalg.det`.

[7]:
```
import time
import pandas as pd
import numpy as np

def control_time_execution(function, matrix, n = 1):
    """ Imprime el tiempo de ejecución promedio de una función dada,␣
 ↪ejecutándola n veces

    Keyword arguments:
    function -- función a medir
    n -- número de iteraciones (default 100)
    """
```

```
    acc = 0
    for _ in range(n):
        start = time.perf_counter() * 1000000
        result = function(matrix)
        end = time.perf_counter() * 1000000
        acc += ((end -start))
    mean_time = acc / n
    print(f"[{function.__name__}] Tiempo de ejecución promedio en {n}␣
 ↪iteraciones: {(mean_time):.2f} microsegundos")
    return result, mean_time


np.random.seed(42)
n_pruebas = 10
min_size = 2
max_size = 10
tolerancia = 1e-6


resultados = []

for i in range(n_pruebas):
    size = np.random.randint(min_size, max_size + 1)
    matrix = np.random.randint(-10, 11, size=(size, size))
    print(f"Test #{i+1} - Tamaño: {size}x{size}")
    print("Matriz:\n", matrix)

    det_recursivo, rec_t = control_time_execution(determinante_recursivo,␣
 ↪matrix)
    det_numpy, np_t = control_time_execution(np.linalg.det, matrix)
    det_gauss, gauss_t = control_time_execution(determinante_gauss, matrix.
 ↪copy())

    resultados.append([
        size,
        det_recursivo,
        det_gauss,
        det_numpy,
        rec_t,
        gauss_t,
        np_t
    ])

    print(f"Determinante (recursivo): {det_recursivo}")
    print(f"Determinante (Gauss):     {det_gauss:.6f}")
    print(f"Determinante (NumPy):     {det_numpy:.6f}")
    if abs(det_recursivo - det_numpy) < tolerancia:
```

```
        print("  Resultado correcto\n")
    else:
        print("  Diferencia significativa\n")
times_execution = pd.DataFrame(
    columns=["Tamaño", "Determinante (Recursivo)", "Determinante (Gauss)",␣
 ↪"Determinante (NumPy)", "Tiempo Recursivo", "Tiempo Gauss", "Tiempo NumPy"],
    data=resultados)

display(times_execution)
```

```
Test #1 - Tamaño: 8x8
Matriz:
 [[  9   4   0  -3  10  -4   8   0]
 [  0  10  -7  -3  -8  10  -9   1]
 [ -5  -9  10 -10   1   1   6  -1]
 [  5   4   4   8   1   9  -8  -6]
 [  8  -4  10  -2  -4   7  -7   3]
 [  7  -2  10  -9   9   4  -4   1]
 [ -3   4  -8   3   6  -7   7  -3]
 [ -7  -9  -5  -1  -7   7   1  -9]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones:
231186.40 microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 86.90 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 232.80
microsegundos
Determinante (recursivo): -17134792.0
Determinante (Gauss):     -17134792.000000
Determinante (NumPy):     -17134792.000000
  Resultado correcto

Test #2 - Tamaño: 5x5
Matriz:
 [[  3   5   4  -3   3]
 [ -3  10   5   2   7]
 [  4  10   2  -2   4]
 [  2 -10  -4  -2 -10]
 [  1  -3   0   8   6]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones: 725.00
microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 15.70 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 110.80
microsegundos
Determinante (recursivo): -7584.0
Determinante (Gauss):     -7584.000000
Determinante (NumPy):     -7584.000000
  Resultado correcto
```

```
Test #3 - Tamaño: 9x9
Matriz:
 [[ -8  -8 -10  -6  -1  -4  -2  -4  -2]
 [ -3   1  -9 -10   5  -6  -8   1  -3]
 [ -8 -10  -8  -6   4   3  -8 -10  -6]
 [  3  -4  -2   4   4  -1   2   8  -4]
 [  6   9  -7  -6  -4   2   4   0  -7]
 [  2  -4   8  -9  -1   2  10  -5   1]
 [  1   9   0  -4 -10 -10   9   2  -2]
 [ -8  -4  -5  -3  -2  -6 -10   8  -1]
 [  1   4  -2   9   6   6   9   1  -4]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones:
2076369.00 microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 63.30 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 275.10
microsegundos
Determinante (recursivo): 4412290442.0
Determinante (Gauss):     4412290442.000001
Determinante (NumPy):     4412290442.000013
 Diferencia significativa

Test #4 - Tamaño: 3x3
Matriz:
 [[-8  6 -6]
 [ 6  6  6]
 [-9 -9 -6]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones: 68.00
microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 29.10 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 353.80
microsegundos
Determinante (recursivo): -252.0
Determinante (Gauss):     -252.000000
Determinante (NumPy):     -252.000000
 Resultado correcto

Test #5 - Tamaño: 2x2
Matriz:
 [[-10   8]
 [ -9  10]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones: 4.60
microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 29.60 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 3.40
microsegundos
Determinante (recursivo): -28.0
Determinante (Gauss):     -28.000000
Determinante (NumPy):     -28.000000
```

```
 Resultado correcto

Test #6 - Tamaño: 7x7
Matriz:
 [[ -7   0   6  -5  -6   9  -9]
 [ -5   0   5   5 -10  -2  -5]
 [  5  -8   9  -7   8  -8   8]
 [  9  -4   9  -2 -10  -3  -4]
 [  7  -3 -10   0   7  -1  -8]
 [ -4   5   5   9   6  -9 -10]
 [  5   1  -6  -6  -2  -2  -8]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones: 28665.20
microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 42.60 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 144.70
microsegundos
Determinante (recursivo): 59302125.0
Determinante (Gauss):     59302125.000000
Determinante (NumPy):     59302125.000000
 Resultado correcto

Test #7 - Tamaño: 4x4
Matriz:
 [[  5   5  -8   9]
 [-10   9   0   6]
 [ -3  -7  -5  -3]
 [  9  -8   5  -8]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones: 153.10
microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 15.60 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 57.80
microsegundos
Determinante (recursivo): -189.0
Determinante (Gauss):     -189.000000
Determinante (NumPy):     -189.000000
 Resultado correcto

Test #8 - Tamaño: 10x10
Matriz:
 [[  7   3   7  -9  -8   5  -2  -7 -10  -7]
 [-10   3  10   5   9  -3  -4  -8   6 -10]
 [  5   1   8   3  -5  -5   2   8  -3  -9]
 [ 10 -10   4 -10  -6   5   8  -7  -8   6]
 [  6   1   3  10  -5  -8  -2  -6   6   3]
 [ 10  -8 -10   9  10 -10  -8   7  -1  -8]
 [ -3   3   7   4  -9  -1  -9   6  -3 -10]
 [ -2   0   5  -4  -1  -8   7   2  -4  -7]
 [  2   9 -10  -3   3   5   3   1   8   4]
```

```
 [ -9  -9   8   6   9  -1  -5   4   0  -6]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones:
20636670.20 microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 48.70 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 258.90
microsegundos
Determinante (recursivo): 23302844006.0
Determinante (Gauss):     23302844006.000004
Determinante (NumPy):     23302844005.999950
 Diferencia significativa

Test #9 - Tamaño: 2x2
Matriz:
 [[-3 10]
 [ 1  1]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones: 4.90
microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 13.00 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 3.70
microsegundos
Determinante (recursivo): -13.0
Determinante (Gauss):     -13.000000
Determinante (NumPy):     -13.000000
 Resultado correcto

Test #10 - Tamaño: 6x6
Matriz:
 [[ -4  -7  -5   2   9   4]
 [ -8  -3   9   5   2   7]
 [ -1   8   6   8  -6  10]
 [ -2   1  10 -10 -10   4]
 [ -9   5  -3   2  10 -10]
 [  5  -4  -6  -8   1   5]]
[determinante_recursivo] Tiempo de ejecución promedio en 1 iteraciones: 4338.90
microsegundos
[det] Tiempo de ejecución promedio en 1 iteraciones: 19.60 microsegundos
[determinante_gauss] Tiempo de ejecución promedio en 1 iteraciones: 102.30
microsegundos
Determinante (recursivo): -858435.0
Determinante (Gauss):     -858435.000000
Determinante (NumPy):     -858435.000000
 Resultado correcto


   Tamaño  Determinante (Recursivo)  Determinante (Gauss)  \
0       8             -1.713479e+07         -1.713479e+07
1       5             -7.584000e+03         -7.584000e+03
2       9              4.412290e+09          4.412290e+09
```

```
3       3             -2.520000e+02         -2.520000e+02
4       2             -2.800000e+01         -2.800000e+01
5       7              5.930212e+07          5.930213e+07
6       4             -1.890000e+02         -1.890000e+02
7      10              2.330284e+10          2.330284e+10
8       2             -1.300000e+01         -1.300000e+01
9       6             -8.584350e+05         -8.584350e+05

   Determinante (NumPy)  Tiempo Recursivo  Tiempo Gauss  Tiempo NumPy
0         -1.713479e+07      2.311864e+05    232.799988     86.900024
1         -7.584000e+03      7.250000e+02    110.799988     15.700012
2          4.412290e+09      2.076369e+06    275.099976     63.299927
3         -2.520000e+02      6.800000e+01    353.800049     29.099976
4         -2.800000e+01      4.600037e+00      3.400024     29.599976
5          5.930212e+07      2.866520e+04    144.700012     42.599976
6         -1.890000e+02      1.531000e+02     57.800049     15.599976
7          2.330284e+10      2.063667e+07    258.899963     48.700012
8         -1.300000e+01      4.899963e+00      3.700012     13.000000
9         -8.584350e+05      4.338900e+03    102.299988     19.599976
```

Interpretación de resultados:

Lo que podemos apreciar es que el algoritmo de eliminación de Gauss ha demostrado ser significativamente más eficiente que el método recursivo para calcular el determinante, especialmente a medida que aumenta el tamaño de la matriz. Esto se debe a que , tal y como analizamos en anteriores apartados el método recursivo tiene una complejidad factorial, lo que lo hace impracticable para matrices de tamaño moderado, mientras que el método de eliminación de Gauss tiene una complejidad cúbica, lo que lo hace mucho más manejable. Adicionalmente, el método más rápido salvo casos muy excepcionales es el de `numpy.linalg.det`, que está optimizado para cálculos numéricos y utiliza algoritmos avanzados para calcular determinantes de manera eficiente, superando incluso al método de eliminación de Gauss en términos de velocidad.

2.a [1 pto] Prográmese en Python el método de descenso de gradiente para funciones de "n" variables. - 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 $\|\nabla f(x)\|_2 < \text{tol}$). - Un número máximo de iteraciones `maxit` (con el fin de evitar ejecuciones indefinidas en caso de divergencia o convergencia muy lenta).

[8]:
```
def gradient_descent(grad_f, initial_v, learnign_rate, tolerance,␣
↪maxiterations=1000):
    point = initial_v if isinstance(initial_v, np.ndarray) else np.
↪array([initial_v], dtype=float)
    is_converged = False
```

```
    for _ in range(int(maxiterations)):
        gradient = grad_f(point)
        #print(f"Gradiente del punto {point}: {gradient}")
        if np.linalg.norm(gradient) < tolerance:
            print(f"Converge en el punto {point} con una tolerancia de␣
 ↪{tolerance} tras realizar {_} iteraciones.")
            is_converged = True
            return point, is_converged, _ + 1

        point -= (learnign_rate * gradient)

    print(f"No ha conseguido converger en {maxiterations} iteraciones.")
    return point, is_converged, maxiterations
```

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

[9]:
```
# Definición de gradiente de anterior función
def one_vairable_grad_f(point):
    return np.array([12*(point[0]**3) + 12*(point[0]**2) - 24*(point[0])],␣
 ↪dtype=float)
```

2.b.I Aplica el método sobre $f(x)$ con $x_0 = 3$ y $\gamma = 0.001$, tol=1e-12, maxit=1e5

[10]:
```
data = []

result, is_converged, n_iterations = gradient_descent(one_vairable_grad_f, 3, 0.
 ↪001, 1e-12 ,10000)
data.append([
    [3],
    result,
    "Si" if is_converged else "No",
    0.001,
    1e-12,
    n_iterations
])
```

```
Converge en el punto [1.] con una tolerancia de 1e-12 tras realizar 831
iteraciones.
```

II [0.5 puntos]Aplica de nuevo el método sobre $f(x)$ con $x_0 = 3$, $\gamma = 0.01$, `tol=1e-12`, `maxit=1e5`.

[11]:
```
result, is_converged, n_iterations= gradient_descent(one_vairable_grad_f, 3, 0.
 ↪01, 1e-12 ,10000)
data.append([
    [3],
    result,
    "Si" if is_converged else "No",
    0.01,
```

```
    1e-12,
    n_iterations
])
```

```
Converge en el punto [-2.] con una tolerancia de 1e-12 tras realizar 31
iteraciones.
```

III [0.5 puntos] Contrasta e interpreta los dos resultados obtenidos en los apartados anteriores y compáralos con los mínimos locales obtenidos analíticamente.

Los resultados obtenidos son los que muestra el siguiente código:

[12]:
```
df_gradient = pd.DataFrame(
    columns=["Punto Inicial", "Resultado", "Convergencia", "Tasa de␣
 ↪Aprendizaje", "Tolerancia", "Iteraciones hasta convergencia"],
    data=data
)

display(df_gradient)
```

```
  Punto Inicial              Resultado Convergencia  Tasa de Aprendizaje  \
0           [3]   [1.0000000000000275]           Si                0.001
1           [3]  [-1.9999999999999882]           Si                0.010

     Tolerancia  Iteraciones hasta convergencia
0  1.000000e-12                             832
1  1.000000e-12                              32
```

Teniendo f: $f(x) = 3x^4 + 4x^3 - 12x^2 + 7.$, sabemos que su $f'(x) = 12x^3 + 12x^2 - 24x$, por lo que los puntos críticos son aquellos que cumplen $f'(x) = 0$, es decir, $12x^3 + 12x^2 - 24x = 0$. Resolviendo esta ecuación, encontramos los puntos críticos y sus correspondientes valores de f. Para ello, sacamos factor común $12x$:

$$12x(x^2 + x - 2) = 0$$

Resolviendo la ecuación cuadrática $x^2 + x - 2 = 0$ con la fórmula general, obtenemos dos soluciones:

$$x_1 = 1, \quad x_2 = -2$$

Por lo tanto, los puntos críticos son $x_1 = 0$, $x_2 = 1$ y $x_3 = -2$. Evaluando la función en estos puntos, obtenemos: - $f(0) = 7$ - $f(1) = 2$ - $f(-2) = 3$

Para saber si son máximos o mínimos locales, calculamos la segunda derivada $f''(x) = 36x^2 + 24x - 24$ y evaluamos en los puntos críticos:

- $f''(0) = -24$ (máximo local)
- $f''(1) = 36$ (mínimo local)
- $f''(-2) = 72$ (mínimo local)

En el caso de la ejecución del método de descenso de gradiente, los dos resultados que ha obtenido [1.0] en la primera ejecución y [-2.0] vemos que se corresponden con los puntos críticos que hemos

obtenido analíticamente que son a su vez mínimos locales, por lo que podemos concluir que el método de descenso de gradiente ha convergido a los mínimos locales de la función.

¿Qué influencia puede llegar a tener la elección del ratio de aprendizaje γ?

La diferencia entre ambos resultados se debe a la elección del ratio de aprendizaje:

- Una tasa de aprendizaje más pequeña hace que el descenso sea más lento pero más preciso, permitiendo caer en el mínimo local más cercano al punto inicial.
- Una tasa de aprendizaje más grande puede mover el punto inicial hacia otro valle antes de estabilizarse, llevándolo a otro mínimo local.

IV [0.5 puntos] Aplica nuevamente el método sobre $f(x)$ con $x_0 = 3$, $\gamma = 0.1$, `tol=1e-12`, `maxit=1e5`. Interpreta el resultado.

[13]:
```
result, is_converged, n_iterations= gradient_descent(one_vairable_grad_f, 3, 0.
↪1, 1e-12 ,10000)
data.append([
    [3],
    result,
    "Si" if is_converged else "No",
    0.1,
    1e-12,
    n_iterations
])
```

```
No ha conseguido converger en 10000 iteraciones.
```

```
C:\Users\maike\AppData\Local\Temp\ipykernel_28192\111289885.py:3:
RuntimeWarning: overflow encountered in scalar power
  return np.array([12*(point[0]**3) + 12*(point[0]**2) - 24*(point[0])],
dtype=float)
C:\Users\maike\AppData\Local\Temp\ipykernel_28192\111289885.py:3:
RuntimeWarning: invalid value encountered in scalar subtract
  return np.array([12*(point[0]**3) + 12*(point[0]**2) - 24*(point[0])],
dtype=float)
```

Interpretación del resultado: Con esta tasa, el método da saltos muy grandes, lo que hace que los valores del gradiente crezcan exponencialmente, y por tanto, el método diverge. Esto indica que la tasa de aprendizaje es demasiado alta para esta función, lo que provoca que el algoritmo no converja a un mínimo local. Incluso nos llegan advertencia de `RuntimeWarning: overflow encountered in scalar power`, lo que indica que los valores se están volviendo demasiado grandes para ser representados correctamente en el tipo de dato utilizado.

V [0.5 puntos] Finalmente, aplica el método sobre $f(x)$ con $x_0 = 0$, $\gamma = 0.001$, `tol=1e-12`, `maxit=1e5`.

[14]:
```
result, is_converged, n_iterations= gradient_descent(one_vairable_grad_f, 0, 0.
↪001, 1e-12 ,10000)

data.append([
```

```
    [0],
    result,
    "Si" if is_converged else "No",
    0.001,
    1e-12,
    n_iterations
])
```

```
Converge en el punto [0.] con una tolerancia de 1e-12 tras realizar 0
iteraciones.
```

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?

el algoritmo **converge inmediatamente en la primera iteración**, detectando que ya cumple con la condición de tolerancia.

Porque el gradiente de la función en $x = 0$ es:

$$f'(x) = 12x^3 + 12x^2 - 24x \quad \Rightarrow \quad f'(0) = 0$$

Esto implica que el algoritmo ya se encuentra en un punto crítico desde el inicio. Por lo tanto, **no necesita realizar ningún paso de actualización**: ya cumple la condición $ | f(x)| < tol $.

¿Es un resultado deseable?

En este caso concreto no, porque en $x = 0$ no tenemos un mínimo local sino un máximo local tal y como ya vimos en el análisis previo.

¿A qué se debe este fenómeno?

Se debe a que el descenso por gradiente solo observa la magnitud del gradiente, no la naturaleza del punto crítico. Al iniciarse en (x = 0), y tener gradiente nulo, asume erróneamente que ha encontrado un punto óptimo.

Conclusión:

Este resultado muestra una **limitación del método** cuando el punto inicial coincide con un punto crítico que **no es un mínimo local**. En estos casos, el análisis analítico previo (como el estudio del signo de la derivada o la convexidad) es clave para validar los resultados numéricos.

0.0.1 Resultados finales

Se muestra una tabla resumen de los resultados obtenidos en el método de descenso de gradiente:

[15]:
```
df_gradient = pd.DataFrame(
    columns=["Punto Inicial", "Resultado", "Convergencia", "Tasa de␣
 ↪Aprendizaje", "Tolerancia", "Iteraciones hasta convergencia"],
    data=data
)

display(df_gradient)
```

```
  Punto Inicial              Resultado Convergencia  Tasa de Aprendizaje  \
0           [3]   [1.0000000000000275]           Si                0.001
1           [3]  [-1.9999999999999882]           Si                0.010
2           [3]                  [nan]           No                0.100
3           [0]                  [0.0]           Si                0.001

     Tolerancia  Iteraciones hasta convergencia
0  1.000000e-12                             832
1  1.000000e-12                              32
2  1.000000e-12                           10000
3  1.000000e-12                               1
```

c) Sea la función $g : \mathbb{R}^2 \to \mathbb{R}$ dada por

$$g(x, y) = x^2 + y^3 + 3xy + 1$$

[16]:
```
def two_variable_grad_f(point):
    """Calcula el gradiente de la función g(x, y) = x^2 + y^3 + 3xy + 1"""
    x, y = point[0], point[1]
    return np.array([
        2*x + 3*y,
        3*(y**2) + 3*x
    ], dtype=float)
```

I [0.5 puntos] Aplíquese el método sobre $g(x, y)$ con $x_0 = (-1, 1)$, $\gamma = 0.01$, `tol=1e-12`, `maxit=1e5`.

[17]:
```
data2 = []

result, is_converged, n_iterations = gradient_descent(two_variable_grad_f,np.
↪array([-1, 1], dtype=float), 0.01, 1e-12 ,10000)

data2.append([
    np.array([-1, 1], dtype=float),
    result,
    "Si" if is_converged else "No",
    0.01,
    1e-12,
    n_iterations
])
```

```
Converge en el punto [-2.25  1.5 ] con una tolerancia de 1e-12 tras realizar
3139 iteraciones.
```

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

En este caso, hasta que no desarrollamos el punto 2.b.III no nos damos cuenta de que el punto (x_0 = (0, 0)) es un punto de silla de la función (g), por lo que el resultado que obtenemos no es deseable, ya que el algoritmo no converge a un mínimo local, sino que se queda en un punto de silla donde el gradiente es cero pero no es un mínimo.

[18]:
```
result, is_converged, n_iterations = gradient_descent(two_variable_grad_f,np.
 ↪array([0,0], dtype=float), 0.01, 1e-12 ,10000)

data2.append([
    np.array([-1, 1], dtype=float),
    result,
    "Si" if is_converged else "No",
    0.01,
    1e-12,
    n_iterations
])
```

```
Converge en el punto [0. 0.] con una tolerancia de 1e-12 tras realizar 0
iteraciones.
```

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

Los resultados obtenidos son los que muestra el siguiente código:

Para el caso de la función $g(x, y) = x^2 + y^3 + 3xy + 1$, tenemos que:

1. El gradiente es:
$$\nabla g(x, y) = \left(\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}\right) = (2x + 3y, 3y^2 + 3x)$$

2. Los puntos críticos se obtienen resolviendo el sistema de ecuaciones:
$$\begin{cases} 2x + 3y = 0 \\ 3y^2 + 3x = 0 \end{cases}$$

De este sistema, obtenemos dos puntos críticos: - ((0, 0)) - ((-2.25, 1.5))

3. El hessiano es:
$$H_g(x, y) = \begin{pmatrix} \frac{\partial^2 g}{\partial x^2} & \frac{\partial^2 g}{\partial x \partial y} \\ \frac{\partial^2 g}{\partial y \partial x} & \frac{\partial^2 g}{\partial y^2} \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 3 & 6y \end{pmatrix}$$

Si aplicamos el punto crítico ((0, 0)):

$$H_g(0, 0) = \begin{pmatrix} 2 & 3 \\ 3 & 0 \end{pmatrix}$$

Si aplicamos el criterio de Sylvester, vemos que el determinante es negativo:

$$\det(H_g(0, 0)) = 2 \cdot 0 - 3 \cdot 3 = -9 < 0$$

Esto indica que ((0, 0)) es un punto de silla, no un mínimo local.

Si aplicamos el punto crítico ((-2.25, 1.5)):

$$H_g(-2.25, 1.5) = \begin{pmatrix} 2 & 3 \\ 3 & 9 \end{pmatrix}$$

$$\det(H_g(-2.25, 1.5)) = 2 \cdot 9 - 3 \cdot 3 = 18 - 9 = 9 > 0$$

Esto indica que ((-2.25, 1.5)) es un mínimo local.

[19]:
```
df_result_multivariable = pd.DataFrame(
    columns=["Punto Inicial", "Resultado", "Convergencia", "Tasa de␣
 ↪Aprendizaje", "Tolerancia", "Iteraciones hasta convergencia"],
    data=data2
)

display(df_result_multivariable)
```

```
  Punto Inicial                                    Resultado Convergencia  \
0   [-1.0, 1.0]  [-2.2499999999989475, 1.4999999999996108]           Si
1   [-1.0, 1.0]                                  [0.0, 0.0]           Si

   Tasa de Aprendizaje    Tolerancia  Iteraciones hasta convergencia
0                 0.01  1.000000e-12                            3140
1                 0.01  1.000000e-12                               1
```

Lo que podemos apreciar es que el algoritmo de descenso de gradiente ha convergido al punto crítico ((-2.25, 1.5)), que es un mínimo local, lo cual es un resultado deseable. Sin embargo cuando usamos como punto inicial (x_0 = (0, 0)), el algoritmo no converge a un mínimo local, sino que se queda en un punto de silla donde el gradiente es cero pero no es un mínimo. Lo cual, como hemos visto, es un resultado no deseable.