

Actividad Guiada 2 de Algoritmos de Optimizacion

Nombre: Roberto Merchán González

Github: <https://github.com/robertomergon/03MIAR-Algoritmos-de-Optimizacion>

```
In [ ]: %pip install matplotlib  
%pip install numpy  
%pip install sympy
```

```
Requirement already satisfied: matplotlib in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (3.9.1)  
Requirement already satisfied: contourpy>=1.0.1 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (1.2.1)  
Requirement already satisfied: cycler>=0.10 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (0.12.1)  
Requirement already satisfied: fonttools>=4.22.0 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (4.53.0)  
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (1.4.5)  
Requirement already satisfied: numpy>=1.23 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (2.0.0)  
Requirement already satisfied: packaging>=20.0 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (2.4.1)  
Requirement already satisfied: pillow>=8 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (10.4.0)  
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (3.1.2)  
Requirement already satisfied: python-dateutil>=2.7 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from matplotlib) (2.9.0)  
Requirement already satisfied: six>=1.5 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)  
Note: you may need to restart the kernel to use updated packages.  
Requirement already satisfied: numpy in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (2.0.0)  
Note: you may need to restart the kernel to use updated packages.  
Requirement already satisfied: sympy in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (1.12.1)  
Requirement already satisfied: mpmath<1.4.0,>=1.1.0 in c:\users\rmerc\documents\github\03miar-algoritmos-de-optimizacion\conda\lib\site-packages (from sympy) (1.3.0)  
Note: you may need to restart the kernel to use updated packages.
```

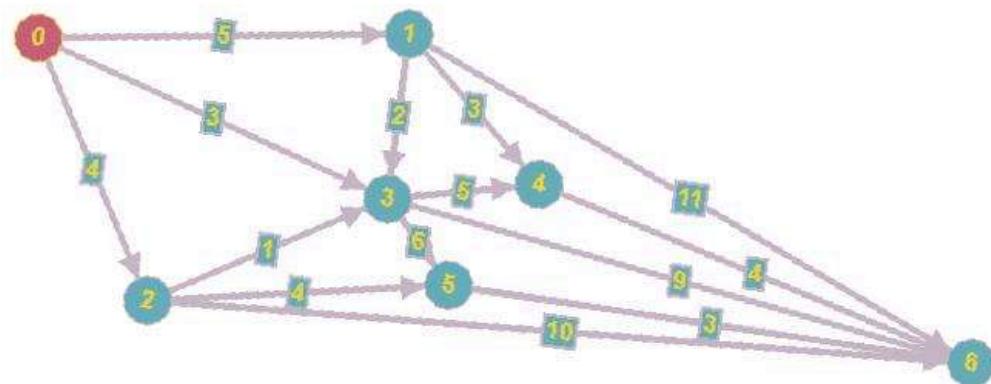
```
In [ ]: import math
```

Ejercicio 1: Programación Dinámica. Viaje por el río

- **Definición:** Es posible dividir el problema en subproblemas más pequeños, guardando las soluciones para ser utilizadas más adelante.
- **Características** que permiten identificar problemas aplicables:
 - Es posible almacenar soluciones de los subproblemas para ser utilizados más adelante
 - Debe verificar el principio de optimalidad de Bellman: "en una secuencia óptima de decisiones, toda sub-secuencia también es óptima" (*)
 - La necesidad de guardar la información acerca de las soluciones parciales unido a la recursividad provoca la necesidad de preocuparnos por la complejidad espacial (cuantos recursos de espacio usaremos)

Problema

En un río hay **n** embarcaderos y debemos desplazarnos río abajo desde un embarcadero a otro. Cada embarcadero tiene precios diferentes para ir de un embarcadero a otro situado más abajo. Para ir del embarcadero i al j , puede ocurrir que sea más barato hacer un trasbordo por un embarcadero intermedio k . El problema consiste en determinar la combinación más barata.



- Consideramos una tabla $\text{TARIFAS}(i,j)$ para almacenar todos los precios que nos ofrecen los embarcaderos.
- Si no es posible ir desde i a j daremos un valor alto para garantizar que ese trayecto no se va a elegir en la ruta óptima(modelado habitual para restricciones)

```
In [ ]: #Viaje por el río - Programación dinámica
```

```
#####
#####
```

```
TARIFAS = [
[0,5,4,3,float("inf"),999,999],  #desde nodo 0
[999,0,999,2,3,999,11],  #desde nodo 1
```

```
[999, 999, 0, 1, 999, 4, 10], #desde nodo 2
[999, 999, 999, 0, 5, 6, 9],
[999, 999, 999, 999, 0, 999, 4],
[999, 999, 999, 999, 999, 0, 3],
[999, 999, 999, 999, 999, 999, 0]
]

#999 se puede sustituir por float("inf") del modulo math
TARIFAS
```

```
Out[ ]: [[0, 5, 4, 3, inf, 999, 999],
[999, 0, 999, 2, 3, 999, 11],
[999, 999, 0, 1, 999, 4, 10],
[999, 999, 999, 0, 5, 6, 9],
[999, 999, 999, 999, 0, 999, 4],
[999, 999, 999, 999, 999, 0, 3],
[999, 999, 999, 999, 999, 999, 0]]
```

```
In [ ]: #Calculo de la matriz de PRECIOS y RUTAS
# PRECIOS - contiene la matriz del mejor precio para ir de un nodo a otro
# RUTAS - contiene los nodos intermedios para ir de un nodo a otro
#####
def Precios(TARIFAS):
#####
    #Total de Nodos
    N = len(TARIFAS[0])

    #Inicialización de la tabla de precios
    PRECIOS = [ [9999]*N for i in [9999]*N] #n x n
    RUTA = [ ["]*N for i in ["]*N]

    #Se recorren todos los nodos con dos bucles(origen - destino)
    # para ir construyendo la matriz de PRECIOS
    for i in range(N-1):
        for j in range(i+1, N):
            MIN = TARIFAS[i][j]
            RUTA[i][j] = i

            for k in range(i, j):
                if PRECIOS[i][k] + TARIFAS[k][j] < MIN:
                    MIN = min(MIN, PRECIOS[i][k] + TARIFAS[k][j])
                RUTA[i][j] = k
            PRECIOS[i][j] = MIN

    return PRECIOS,RUTA
```

```
In [ ]: PRECIOS,RUTA = Precios(TARIFAS)
#print(PRECIOS[0][6])

print("PRECIOS")
for i in range(len(TARIFAS)):
    print(PRECIOS[i])

print("\nRUTA")
for i in range(len(TARIFAS)):
    print(RUTA[i])
```

PRECIOS

```
[9999, 5, 4, 3, 8, 8, 11]
[9999, 9999, 999, 2, 3, 8, 7]
[9999, 9999, 9999, 1, 6, 4, 7]
[9999, 9999, 9999, 9999, 5, 6, 9]
[9999, 9999, 9999, 9999, 9999, 999, 4]
[9999, 9999, 9999, 9999, 9999, 9999, 3]
[9999, 9999, 9999, 9999, 9999, 9999]
```

RUTA

```
['', 0, 0, 0, 1, 2, 5]
 ['', '', 1, 1, 1, 3, 4]
 ['', '', '', 2, 3, 2, 5]
 ['', '', '', '', 3, 3, 3]
 ['', '', '', '', '', 4, 4]
 ['', '', '', '', '', '', 5]
 ['', '', '', '', '', '', '']
```

In []: #Calculo de la ruta usando la matriz RUTA

```
def calcular_ruta(RUTA, desde, hasta):
    if desde == RUTA[desde][hasta]:
        #if desde == hasta:
        #print("Ir a :" + str(desde))
        return desde
    else:
        return str(calcular_ruta(RUTA, desde, RUTA[desde][hasta])) + ',' + str(RUTA[hasta])
print("\nLa ruta es:")
calcular_ruta(RUTA, 0,6)
```

La ruta es:

Out[]: '0,2,5'

Ejercicio 2: Problema de Asignacion de tarea

In []: #Asignacion de tareas - Ramificación y Poda

```
#####
#      T A R E A
#      A
#      G
#      E
#      N
#      T
#      E

COSTES=[[11,12,18,40],
        [14,15,13,22],
        [11,17,19,23],
        [17,14,20,28]]
```

In []: #Calculo del valor de una solucion parcial

```
def valor(S,COSTES):
    VALOR = 0
    for i in range(len(S)):
        VALOR += COSTES[S[i]][i]
    return VALOR

valor((3,2, ),COSTES)
```

Out[]: 34

```
In [ ]: #Coste inferior para soluciones parciales
# (1,3,) Se asigna la tarea 1 al agente 0 y la tarea 3 al agente 1

def CI(S,COSTES):
    VALOR = 0
    #Valores establecidos
    for i in range(len(S)):
        VALOR += COSTES[i][S[i]]

    #Estimacion
    for i in range( len(S), len(COSTES) ):
        VALOR += min( [ COSTES[j][i] for j in range(len(S), len(COSTES)) ] )
    return VALOR

def CS(S,COSTES):
    VALOR = 0
    #Valores establecidos
    for i in range(len(S)):
        VALOR += COSTES[i][S[i]]

    #Estimacion
    for i in range( len(S), len(COSTES) ):
        VALOR += max( [ COSTES[j][i] for j in range(len(S), len(COSTES)) ] )
    return VALOR

CI((0,1),COSTES)
```

Out[]: 68

```
In [ ]: #Genera tantos hijos como posibilidades haya para la siguiente elemento de
#(0,) -> (0,1), (0,2), (0,3)
def crear_hijos(NODO, N):
    HIJOS = []
    for i in range(N):
        if i not in NODO:
            HIJOS.append({'s':NODO +(i,)})
    return HIJOS
```

In []: crear_hijos((0,), 4)

Out[]: [{ 's': (0, 1)}, { 's': (0, 2)}, { 's': (0, 3)}]

```
In [ ]: def ramificacion_y_poda(COSTES):
#Construccion iterativa de soluciones(arbol). En cada etapa asignamos un agente(
#Nodos del grafo { s:(1,2),CI:3,CS:5 }
#print(COSTES)
DIMENSION = len(COSTES)
MEJOR_SOLUCION=tuple( i for i in range(len(COSTES)) )
CotaSup = valor(MEJOR_SOLUCION,COSTES)
#print("Cota Superior:", CotaSup)

NODOS=[]
NODOS.append({ 's':(), 'ci':CI((),COSTES) })
iteracion = 0
```

```

while( len(NODOS) > 0):
    iteracion +=1

    nodo_prometedor = [ min(NODOS, key=lambda x:x['ci']) ][0]['s']
    #print("Nodo prometedor:", nodo_prometedor)

    #Ramificacion
    #Se generan los hijos
    HIJOS =[ { 's':x['s'], 'ci':CI(x['s'], COSTES) } for x in crear_hijos(nodo_]

    #Revisamos la cota superior y nos quedamos con la mejor solucion si llegamos
    NODO_FINAL = [x for x in HIJOS if len(x['s']) == DIMENSION ]
    if len(NODO_FINAL ) >0:
        #print("\n*****Soluciones:", [x for x in HIJOS if len(x['s']) == DIMEN
        if NODO_FINAL[0]['ci'] < CotaSup:
            CotaSup = NODO_FINAL[0]['ci']
            MEJOR_SOLUCION = NODO_FINAL

    #Poda
    HIJOS = [x for x in HIJOS if x['ci'] < CotaSup ]

    #Añadimos los hijos
    NODOS.extend(HIJOS)

    #Eliminamos el nodo ramificado
    NODOS = [ x for x in NODOS if x['s'] != nodo_prometedor ]

print("La solucion final es:" ,MEJOR_SOLUCION , " en " , iteracion , " iteraci

```

ramificacion_y_poda(COSTES)

La solucion final es: [{ 's': (1, 2, 0, 3), 'ci': 64}] en 10 iteraciones para dimension: 4

Ejercicio opcional:

Análisis para mejorar nota:

- Generar matrices con valores aleatorios de mayores dimensiones (5,6,7,...) y ejecutar ambos algoritmos.
- ¿A partir de que dimensión el algoritmo por fuerza bruta deja de ser una opción?
- ¿Hay algún valor de la dimensión a partir de la cual el algoritmo de ramificación y poda también deja de ser una opción válida?

```
In [ ]: import numpy as np

def generar_matriz_aleatoria(dim):
    return np.random.randint(1, 100, size=(dim, dim)).tolist()
```

```
In [ ]: import itertools

def fuerza_bruta(COSTES):
    DIMENSION = len(COSTES)
    permutaciones = list(itertools.permutations(range(DIMENSION)))
    MEJOR_SOLUCION = permutaciones[0]
```

```
CotaSup = valor(MEJOR_SOLUCION, COSTES)

for perm in permutaciones:
    costo = valor(perm, COSTES)
    if costo < CotaSup:
        CotaSup = costo
        MEJOR_SOLUCION = perm

print("La mejor solución es:", MEJOR_SOLUCION, "con un costo de:", CotaSup)
```

```
In [ ]: import time

def evaluar_algoritmo(dimensiones, algoritmo):
    for dim in dimensiones:
        matriz = generar_matriz_aleatoria(dim)
        start_time = time.time()
        algoritmo(matriz)
        end_time = time.time()
        print(f"Dimensión: {dim}, Tiempo de ejecución: {end_time - start_time:.4f} segundos")

dimensiones_a_evaluar = [5, 6, 7, 8, 9, 10]
print("Evaluación de fuerza bruta")
evaluar_algoritmo(dimensiones_a_evaluar, fuerza_bruta)
print("\nEvaluación de ramificación y poda")
evaluar_algoritmo(dimensiones_a_evaluar, ramificacion_y_poda)
```

Evaluación de fuerza bruta

```
La mejor solución es: (4, 1, 2, 0, 3) con un costo de: 106
Dimensión: 5, Tiempo de ejecución: 0.0000 segundos
La mejor solución es: (5, 3, 0, 4, 1, 2) con un costo de: 105
Dimensión: 6, Tiempo de ejecución: 0.0000 segundos
La mejor solución es: (0, 1, 4, 3, 2, 6, 5) con un costo de: 146
Dimensión: 7, Tiempo de ejecución: 0.0080 segundos
La mejor solución es: (2, 5, 0, 1, 7, 6, 4, 3) con un costo de: 110
Dimensión: 8, Tiempo de ejecución: 0.0675 segundos
La mejor solución es: (4, 8, 1, 7, 6, 2, 0, 3, 5) con un costo de: 81
Dimensión: 9, Tiempo de ejecución: 0.5129 segundos
La mejor solución es: (8, 6, 1, 2, 0, 7, 5, 3, 9, 4) con un costo de: 165
Dimensión: 10, Tiempo de ejecución: 5.3842 segundos
```

Evaluación de ramificación y poda

```
La solucion final es: [{"s": (3, 2, 0, 1, 4), "ci": 128}] en 20 iteraciones para dimension: 5
Dimensión: 5, Tiempo de ejecución: 0.0000 segundos
La solucion final es: [{"s": (4, 2, 0, 3, 5, 1), "ci": 162}] en 132 iteraciones para dimension: 6
Dimensión: 6, Tiempo de ejecución: 0.0000 segundos
La solucion final es: [{"s": (2, 1, 5, 0, 4, 3, 6), "ci": 114}] en 157 iteraciones para dimension: 7
Dimensión: 7, Tiempo de ejecución: 0.0088 segundos
La solucion final es: [{"s": (7, 0, 1, 2, 4, 3, 5, 6), "ci": 147}] en 951 iteraciones para dimension: 8
Dimensión: 8, Tiempo de ejecución: 0.0754 segundos
La solucion final es: [{"s": (4, 6, 0, 3, 5, 2, 7, 1, 8), "ci": 154}] en 2295 iteraciones para dimension: 9
Dimensión: 9, Tiempo de ejecución: 0.5247 segundos
La solucion final es: [{"s": (3, 1, 9, 7, 0, 2, 4, 6, 8, 5), "ci": 122}] en 253 iteraciones para dimension: 10
Dimensión: 10, Tiempo de ejecución: 0.0268 segundos
```

Dado que la complejidad del algoritmo de fuerza bruta es de $O(n!)$, el algoritmo deja de ser una opción a partir de $n=10$. Por otro lado, el algoritmo de ramificación y poda es de $O(n^2)$, por lo que es una opción válida para $n=10$.

El algoritmo de ramificación y poda puede ser más eficiente que el de fuerza bruta en la mayoría de los casos, pero no siempre. Este puede dejar de ser eficiente si la cantidad de tareas es muy grande, el número de tareas límite dependerá de la heurística utilizada y de la cantidad de recursos disponibles. Por lo tanto, no se puede determinar un valor exacto de la dimensión a partir de la cual el algoritmo de ramificación y poda deja de ser una opción válida.

Ejercicio 3: Descenso del gradiente

```
In [ ]: import math                      #Funciones matematicas
import matplotlib.pyplot as plt         #Generacion de graficos (otra opcion seaborn)
import numpy as np                      #Tratamiento matriz N-dimensionales y otras (fu
# import scipy as sc

import random
```

Vamos a buscar el minimo de la función paraboloide :

$$f(x) = x^2 + y^2$$

Obviamente se encuentra en $(x,y)=(0,0)$ pero probaremos como llegamos a él a través del descenso del gradiente.

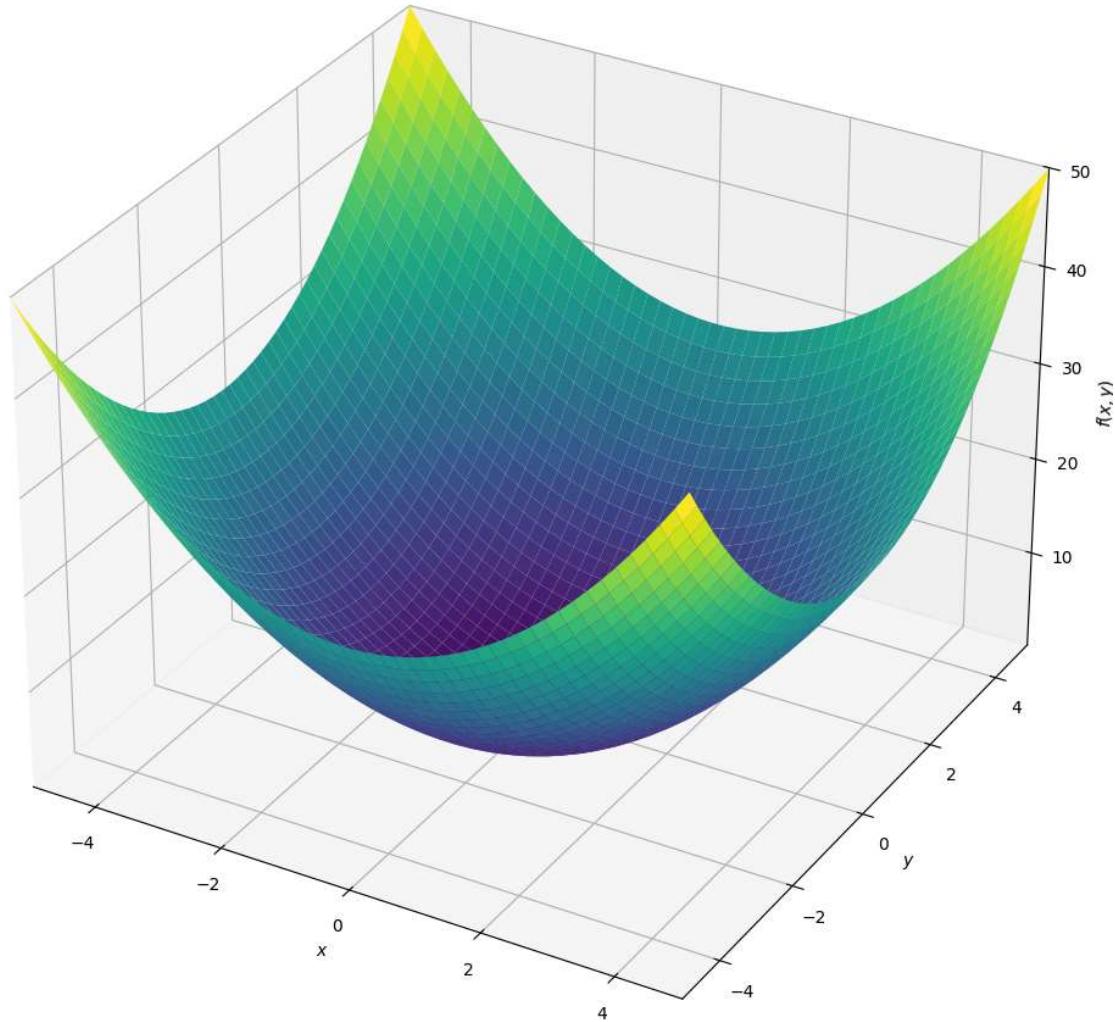
```
In [ ]: #Definimos la funcion
#Paraboloide
f = lambda X: X[0]**2 + X[1]**2      #Funcion
df = lambda X: [2*X[0] , 2*X[1]]       #Gradiente

df([1,2])
```

Out[]: [2, 4]

```
In [ ]: from sympy import symbols
from sympy.plotting import plot
from sympy.plotting import plot3d
x,y = symbols('x y')
plot3d(x**2 + y**2,
       (x,-5,5),(y,-5,5),
       title='x**2 + y**2',
       size=(10,10))
```

$$x^{**2} + y^{**2}$$



Out[]: <sympy.plotting.plot.Plot at 0x278507a4a50>

```
In [ ]: #Prepara los datos para dibujar mapa de niveles de Z
resolucion = 100
rango=5.5

X=np.linspace(-rango,rango,resolucion)
Y=np.linspace(-rango,rango,resolucion)
Z=np.zeros((resolucion,resolucion))
for ix,x in enumerate(X):
    for iy,y in enumerate(Y):
        Z[iy,ix] = f([x,y])

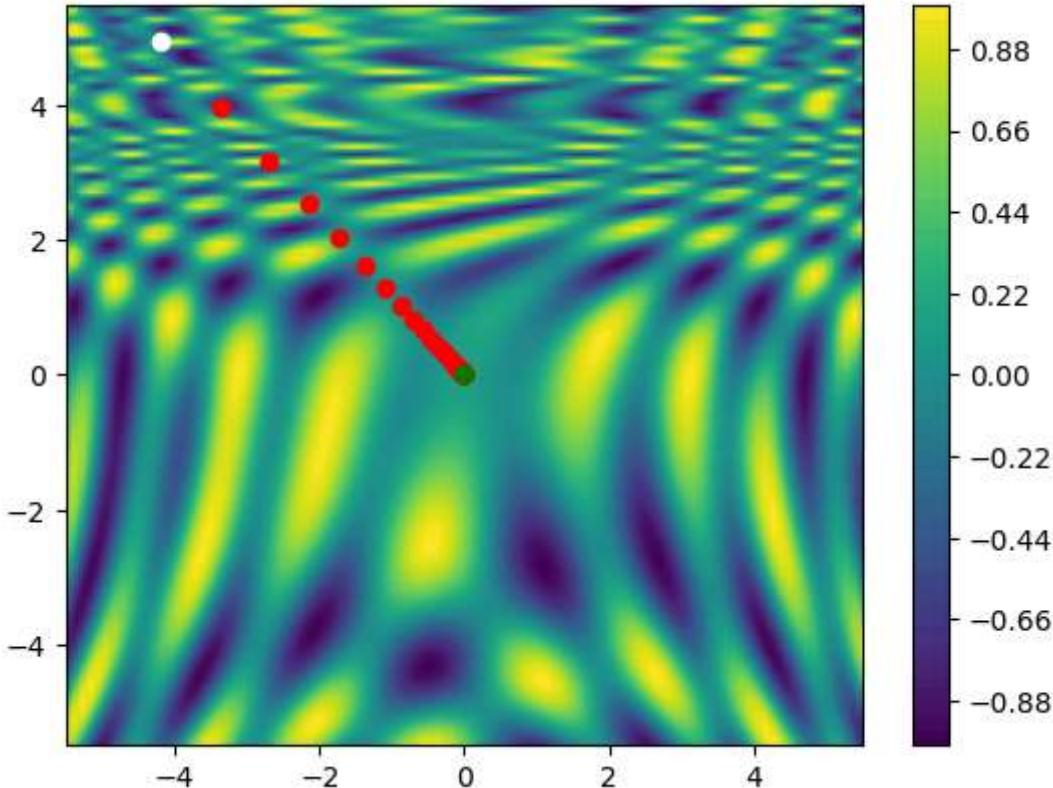
#Pinta el mapa de niveles de Z
plt.contourf(X,Y,Z,resolucion)
plt.colorbar()

#Generamos un punto aleatorio inicial y pintamos de blanco
P=[random.uniform(-5,5 ),random.uniform(-5,5 ) ]
plt.plot(P[0],P[1],"o",c="white")

#Tasa de aprendizaje. Fija. Sería más efectivo reducirlo a medida que nos acerca
TA=.1
```

```
#Iteraciones:50
for _ in range(50):
    grad = df(P)
    #print(P,grad)
    P[0],P[1] = P[0] - TA*grad[0] , P[1] - TA*grad[1]
    plt.plot(P[0],P[1],"o",c="red")

#Dibujamos el punto final y pintamos de verde
plt.plot(P[0],P[1],"o",c="green")
plt.show()
print("Solucion:", P , f(P))
```



Solucion: [-5.997773337346357e-05, 7.06613408313909e-05] 0.14112000495111882

¿Te atreves a optimizar la función?:

$$f(x) = \sin(1/2 * x^2 - 1/4 * y^2 + 3) * \cos(2 * x + 1 - e^y)$$

```
In [ ]: #Definimos la funcion
f= lambda X: math.sin(1/2 * X[0]**2 - 1/4 * X[1]**2 + 3) *math.cos(2*X[0] + 1 -

def gradiente_f(X):
    h = 1e-5 # Un pequeño cambio en las variables para la diferenciación numérica
    grad = np.zeros_like(X)
    for i in range(len(X)):
        X_plus_h = X.copy()
        X_plus_h[i] += h
        X_minus_h = X.copy()
        X_minus_h[i] -= h
        grad[i] = (f(X_plus_h) - f(X_minus_h)) / (2 * h)
    return grad

def descenso_gradiente(f, grad_f, initial_X, ratio_aprendizaje=0.0001, max_iter=1000):
    X = np.array(initial_X)
    for i in range(max_iter):
        grad = grad_f(X)
        if np.linalg.norm(grad) < ratio_aprendizaje:
            break
        X -= ratio_aprendizaje * grad
```

```

for i in range(max_iter):
    grad = grad_f(X)
    X_new = X - ratio_aprendizaje * grad
    if np.linalg.norm(X_new - X) < tolerancia:
        break
    X = X_new
return X

X0 = [1.0, 1.0]
X_opt = descenso_gradiente(f, gradiente_f, X0)
print("Solución:", X_opt, f(X_opt))
print("Valor en el punto inicial:", f(X0))

```

Solución: [2.19866526 1.68559549] -0.9999827469785204
 Valor en el punto inicial: -0.10392999507424829

Con un ratio de aprendizaje lo suficientemente pequeño y un número de iteraciones lo suficientemente amplio como para que el algoritmo converja, se puede llegar al mínimo de la función. Sin embargo, si el ratio de aprendizaje es muy grande, el algoritmo puede no converger y diverger.

Por otro lado, si el número de iteraciones es muy pequeño, el algoritmo puede no converger y no llegar al mínimo de la función.

También hay que tener una buena elección del punto de inicio, ya que si el punto de inicio es muy lejano del mínimo de la función, el algoritmo puede no converger y no llegar al mínimo de la función o si el punto de inicio es un punto crítico, el algoritmo puede no converger y no llegar al mínimo de la función.

En este caso vemos como el valor en el punto inicial es mayor que el valor en el punto final, por lo que podemos decir que el algoritmo ha convergido hacia un mínimo local. Por lo tanto, el algoritmo se ha comportado correctamente.