Práctica Calificada N^{o} 2

Arom Alexander Avila Chumbiauca - 19140097
Adrian Marcel Villafan Virhuez Arom - 20140122

Evaluación continua realizada para el curso de
Optimización

Escuela Profesional de Computación Científica
Facultad de Ciencias Matemáticas
Universidad Nacional Mayor de San Marcos (UNMSM)

2024

Ejercicio 1

Grafique las curvas de nivel de la función

$$f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$$

Se desarrolló el siguiente código en Python se encarga de realizar el gráfico:

```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Definimos las variables simbolicas
x1, x2 = sp.symbols('x1 x2')

# Definimos la funcion a optimizar
opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))

# Convertimos la funcion sympy a una funcion numpy para evaluacion eficiente
opti_np = sp.lambdify((x1, x2), opti, 'numpy')

# Creamos una cuadricula de valores para x1 y x2
x1_vals = np.linspace(-2, 2, 100)
x2_vals = np.linspace(-2, 2, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluamos la funcion en cada punto de la cuadricula
Z = opti_np(X1, X2)

# Creamos la grafica de las curvas de nivel
plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, Z, levels=20, cmap='viridis')
```

```
plt.colorbar(contour)
plt.title('Curvas de Nivel de la Funcion $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.grid(True)
plt.show()
```

Listing 1: Código para graficar las curvas de nivel de la función

Explicación del Código

- `import sympy as sp`, `numpy as np`, `matplotlib.pyplot as plt`: Importación de bibliotecas necesarias para trabajar con funciones simbólicas (`sympy`), manejar datos numéricos (`numpy`) y crear gráficos (`matplotlib`).
- `x1, x2 = sp.symbols('x1 x2')`: Se definen las variables simbólicas x_1 y x_2 usando `sympy(sp)`.
- `opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))`: Se define la función a optimizar, $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$, en variable simbólica.
- `opti_np = sp.lambdify((x1, x2), opti, 'numpy')`: Convertimos la función simbólica a una función que puede ser evaluada numéricamente utilizando `numpy`.
- `x1_vals = np.linspace(-2, 2, 100)` `x2_vals = np.linspace(-2, 2, 100)`: Creamos un rango de valores para x_1 y x_2 desde -2 hasta 2 con 100 puntos en cada dirección.
- `X1, X2 = np.meshgrid(x1_vals, x2_vals)`: Creamos una cuadrícula de valores de x_1 y x_2 utilizando `meshgrid`, lo que permite evaluar la función en una rejilla de puntos.
- `Z = opti_np(x1, x2)`: Evaluamos la función $f(x_1, x_2)$ en cada punto de la cuadrícula.

- `plt.figure(figsize=(8, 6))`: Configuramos el tamaño de la figura para la gráfica.
- `contour = plt.contour(X1, X2, Z, levels=20, cmap='viridis')`: Creamos las curvas de nivel de la función utilizando `contour`, con 20 niveles y una paleta de colores `viridis`.
- `plt.colorbar(contour)`: Añadimos una barra de colores a la gráfica para indicar los valores de las curvas de nivel.
- `plt.title('Curvas de Nivel de la Función` $-e^{-(x_1^2+x_2^2)}$`')`: Añadimos un título a la gráfica.
- `plt.xlabel('`x_1`') plt.ylabel('`x_2`')`: Etiquetamos los ejes x_1 y x_2.
- `plt.grid(True)`: Añadimos una rejilla a la gráfica para facilitar la visualización.
- `plt.show()`: Mostramos la gráfica.

Explicación Matemática

La función $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$ es una función que toma valores negativos debido al factor $-e$. Esta función tiene un mínimo global en el punto $(0, 0)$, donde su valor es -1. Las curvas de nivel representan los lugares geométricos donde la función toma valores constantes. En este caso, las curvas de nivel se forman alrededor del mínimo global, y muestran cómo los valores de la función cambian en el plano x_1, x_2.

Ejercicio 2

$$f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$$

Cuasiconvexo

Usando la definición:

$$f(\lambda x + (\rho - \lambda)y) \leq \max\{f(x), f(y)\} \quad \lambda \in [0, 1]$$

Demostremos que para el par de puntos (x_1, y_1) y (x_2, y_2) se cumple:

$$f(\lambda x_1 + (1 - \lambda)x_2, \lambda y_1 + (1 - \lambda)y_2) \leq \max\{f(x), f(y)\}$$

Donde, al evaluar en f, debemos demostrar:

$$-e^{-(\lambda x_1+(1-\lambda)x_2)^2-(\lambda y_1+(1-\lambda)y_2)^2} \leq \max\{-e^{-(x_1^2+y_1^2)}, -e^{-(x_2^2+y_2^2)}\}$$

Tomando $f(x_1, y_1) \geq f(x_2, y_2) \to \max\{f(x_1, y_1), f(x_2, y_2)\} = f(x_1, y_1)$, por lo que debemos demostrar:

$$-e^{-(\lambda x_1+(1-\lambda)x_2)^2-(\lambda y_1+(1-\lambda)y_2)^2} \leq -e^{-(x_1^2+y_1^2)}$$

Empleamos la desigualdad de Jensen para la función convexa $g(x) = t^2$ función cuadrática donde:

$$(\lambda x_1 + (1 - \lambda x_2))^2 \leq \lambda x_1^2 + (1 - \lambda) x_2^2$$

$$(\lambda y_1 + (1 - \lambda y_2))^2 \leq \lambda y_1^2 + (1 - \lambda) y_2^2$$

Al sumarlos tenemos:

$$Z = (\lambda x_1 + (1 - \lambda x_2))^2 + (\lambda y_1 + (1 - \lambda y_2))^2 \leq \lambda x_1^2 + (1 - \lambda) x_2^2 + \lambda y_1^2 + (1 - \lambda) y_2^2$$

Esto muestra que Z es menor o igual a la convención convexa de los exponentes de los puntos (x_1, y_1) y (x_2, y_2) y puesto que $f(x_1, y_1) \geq f(x_2, y_2)$, tenemos:

$$-e^{-(\lambda x_1 + (1-\lambda)x_2)^2 - (\lambda y_1 + (1-\lambda) y_2)^2} \leq -e^{-(x_1^2 + y_1^2)}$$

Cumpliendo con:

$$f(\lambda x_1 + (1 - \lambda) x_2, \lambda y_1 + (1 - \lambda) y_2) \leq \max\{f(x_1, y_1), f(x_2, y_2)\} = f(x_1, y_1)$$

$$\therefore\ f(x_1, x_2) = -e^{-(x_1^2 + x_2^2)} \text{ es cuasiconvexa}$$

Convexa

Definición de convexidad para las puntas $(x_1, y_1) \quad (x_2, y_2)$:

$$f(\lambda x_1 + (1 - \lambda) x_2, \lambda y_1 + (1 - \lambda) y_2) \leq \lambda f(x_1, y_1) + (1 - \lambda) f(x_2, y_2)$$

$$\lambda \in [0, 1]$$

Contra ejemplo de no convexo

Consideramos $(x_1, y_1) = (0, 0) \quad (x_2, y_2) = (2, 0) \quad \lambda = 0.5$:

$$\begin{aligned}
\lambda f(x_1, y_1) + (1-\lambda) f(x_2, y_2) &= 0.5 f(0,0) + 0.5 f(2,0) \\
&= 0.5(-e^0) + 0.5(-e^{-4}) \\
&= -0.5(1) + 0.5(-e^{-4}) \\
&= -0.5 - 0.5e^{-4} \\
&= -0.50916
\end{aligned}$$

$$\begin{aligned}
f(\lambda x_1 + (1-\lambda)x_2, \lambda y_1 + (1-\lambda)y_2) &= f(0.5(2), 0) \\
&= f(1, 0) \\
&= -e^{-1} \\
&= -0.36788
\end{aligned}$$

Donde:

$$f(\lambda x_1 + (1-\lambda)x_2, \lambda y_1 + (1-\lambda)y_2) \leq \lambda f(x_1, y_1) + (1-\lambda) f(x_2, y_2)$$

$$-0.3678 \leq -0.50916 \quad (\text{¡absurdo!})$$

$\therefore$ La función $f(x_1, x_2) = -e^{(x_1+x_2)}$ no es convexa.

Ejercicio 3

Minimizar $f(x_1, x_2) = -e^{-(x_1^2 - x_2^2)}$

Sujeto a:

$$(x_1, x_2) \in \mathbb{R}^2$$

Hallamos puntos críticos

$$\nabla f(x_1, x_2) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix} = 0$$

$$\frac{\partial f}{\partial x_1} = (2x_1^{e^{-(x_1^2+x_2^2)}}) = 0$$
$$\frac{\partial f}{\partial x_2} = (2x_2^{e^{-(x_1^2+x_2^2)}}) = 0$$

Como $e^{-(x_1^2+x_2^2)}$ no puede ser 0:

$$x_1 = 0, \quad x_2 = 0$$

$$x^* = (0, 0)$$

Hallamos la Hessiana

$$Hf(x_1, x_2) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{pmatrix}$$

$$\frac{\partial^2 f}{\partial x_1^2} = (2 - 4x_1^2 e^{-(x_1^2+x_2^2)})$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = (-4x_1 x_2 e^{-(x_1^2+x_2^2)})$$

$$\frac{\partial^2 f}{\partial x_2^2} = (2 - 4x_2^2 e^{-(x_1^2+x_2^2)})$$

$$Hf(x_1, x_2) = \begin{pmatrix} (2-4x_1^2)e^{-(x_1^2+x_2^2)} & (-4x_1x_2)e^{-(x_1^2+x_2^2)} \\ (-4x_1x_2)e^{-(x_1^2+x_2^2)} & (2-4x_2^2)e^{-(x_1^2+x_2^2)} \end{pmatrix}$$

Evaluando en $x^* = (0, 0)$:

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

$$H(0,0) \succ 0 \quad \text{(Definida positiva)}$$

$$\Rightarrow x^* = (0,0) \text{ es mínimo local}$$

Considerando que $(x_1, x_2) \to \infty \wedge e^{-(x_1^2+x_2^2)} = 0$:

$$f(0,0) = -1 \quad \text{entonces} \quad x^* = (0,0) \quad \text{además de mínimo local es mínimo global}$$

Ejercicio 4

$$f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$$

Vamos a aplicar el método del gradiente con búsqueda exacta y de Armijo. A continuación, se presenta el código y su explicación.

Método de Búsqueda Exacta

```python
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

# Definimos las variables simbolicas
x1, x2 = sp.symbols('x1 x2')

# Definimos la funcion a optimizar
opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))

# Variables iniciales
x0 = np.array([-1.2, 1], dtype=np.float64)
k = 1
error = 1e-5

# Lista que almacenara los datos de cada iteracion
data = []

def busqueda_exacta(x0, opti, grad):
    alpha = sp.symbols('alpha')
    x_new = x0 - alpha * grad
    func_new = opti.subs({x1: x_new[0], x2: x_new[1]})
    dfunc_new = sp.diff(func_new, alpha)
```

```python
    # Encontrar la solucion de alpha que minimiza la funcion
    alpha_opt = sp.solve(dfunc_new, alpha)
    alpha_opt = [sol.evalf() for sol in alpha_opt if sol.is_real and sol > 0]

    if alpha_opt:
        return min(alpha_opt)
    else:
        return 1  # Valor de fallback en caso de que no se encuentre un valor optimo

# Iniciar el contador de tiempo
start_time = time.time()

while k < 101:
    # Obtener el gradiente de la funcion
    grad = np.array([float(opti.diff(var).subs({x1: x0[0], x2: x0[1]})) for var in (
        x1, x2)])

    # Obtener la norma del gradiente
    Normadf = np.linalg.norm(grad)

    # Condicion de terminacion del algoritmo
    if Normadf < error:
        break

    # Obtener el valor optimo de alpha mediante busqueda exacta
    alpha = busqueda_exacta(x0, opti, grad)

    # Actualizar el valor de x0
    x0 = x0 - float(alpha) * grad

    # Calcular el valor de la funcion en x0
    fun = float(opti.subs({x1: x0[0], x2: x0[1]}))

    # Calcular el error
    mod = np.linalg.norm(x0 - np.array([0, 0]))

    # Datos de la iteracion actual a la lista
    data.append([k, float(alpha), x0.tolist(), fun, mod])

    k += 1

# Crear un DataFrame de pandas con los datos recolectados
df = pd.DataFrame(data, columns=['Iteracion', 'Alpha', 'x0', 'f(x0)', '||x0 - x||'])
```

```python
# Imprimir la tabla
print(df)

# Imprimir el tiempo de ejecucion del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecucion: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-2, 2, 100)
x2_vals = np.linspace(-2, 2, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la funcion en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))

# Crear el grafico de las curvas de nivel
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Funcion $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos
    $x_0$')

# Leyenda
plt.legend()

# Mostrar la grafica
plt.show()
```

Listing 2: Código para método de búsqueda exacta

Explicación del Código

- `import sympy as sp, numpy as np, pandas as pd, matplotlib.pyplot as plt, time`: Importamos las bibliotecas necesarias para trabajar con funciones simbólicas (`sympy`), manejar datos numéricos (`numpy`), crear gráficos (`matplotlib`) y medir el tiempo de ejecución (`time`).

- `x1, x2 = sp.symbols('x1 x2')`: Definimos las variables simbólicas x_1 y x_2 usando `sympy`.
- `opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))`: Definimos la función a optimizar, $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$.
- `x0 = np.array([-1.2, 1], dtype=np.float64)`: Establecemos el punto inicial x_0.
- `data = []`: Inicializamos una lista para almacenar los datos de cada iteración.
- `def busqueda_exacta(x0, opti, grad)`: Definimos la función para realizar la búsqueda exacta del paso óptimo α.
- `while k < 101`: Iniciamos el bucle de iteraciones del método del gradiente.
- `grad = np.array([float(opti.diff(var).subs(x1: x0[0], x2: x0[1])) for var in (x1, x2)])`: Calculamos el gradiente de la función en el punto x_0.
- `Normadf = np.linalg.norm(grad)`: Calculamos la norma del gradiente.
- `if Normadf < error: break`: Condición de terminación del algoritmo.
- `alpha = busqueda_exacta(x0, opti, grad)`: Obtenemos el valor óptimo de α mediante búsqueda exacta.
- `x0 = x0 - float(alpha) * grad`: Actualizamos el valor de x_0.
- `fun = float(opti.subs(x1: x0[0], x2: x0[1]))`: Calculamos el valor de la función en x_0.
- `mod = np.linalg.norm(x0 - np.array([0, 0]))`: Calculamos el error.
- `data.append([k, float(alpha), x0.tolist(), fun, mod])`: Añadimos los datos de la iteración actual a la lista.
- `df = pd.DataFrame(data, columns=['Iteración', 'Alpha', 'x0', 'f(x0)',`

`'||x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados.

- `print(df)`: Imprimimos la tabla.
- `execution_time = time.time() - start_time`: Calculamos el tiempo de ejecución del programa.
- `plt.contour(X1, X2, Z, levels=20)`: Creamos el gráfico de las curvas de nivel.
- `plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos` x_0`')`: Extraemos y graficamos los puntos x_0 del DataFrame.
- `plt.show()`: Mostramos la gráfica.

Método de Búsqueda de Armijo

```
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

# Definimos el polinomio a optimizar
x1, x2 = sp.symbols('x1 x2')
opti = -sp.exp(-((x1 ** 2) + (x2 ** 2)))

# Variables iniciales
x0 = np.array([-1.2, 1], dtype=np.float64)
k = 1
error = 1e-5

data = []

def armijo(x0, opti):
  b, s, o, k1, m = 0.5, 1, 0.1, 1, 0

  while True:
    # Hallamos el valor de lambda
    lmb = s * (b ** m)
    # Hallamos f(x0)
    f = float(opti.subs(zip((x1, x2), x0)))
```

```python
    # Hallamos el valor de la gradiente
    grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1,
        x2)])
    # Hallamos el valor de f(x0 - lmb * grad)
    f_k = float(opti.subs(zip((x1, x2), x0 - lmb * grad)))
    mod = grad[0] ** 2 + grad[1] ** 2

    # Condicion de parada
    if f_k <= f - o * lmb * mod:
      break
    else:
      m += 1
      k1 += 1

  return k1, lmb

# Iniciar el contador de tiempo
start_time = time.time()

while k < 101:
  # Obtener la gradiente de la funcion
  grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1, x2)
      ])

  # Obtener la norma del gradiente
  Normadf = (grad[0]**2 + grad[1]**2)**(0.5)

  # Condicion para finalizar el programa
  if Normadf < error:
    break

  # Obtener el numero de iteraciones internas y el valor de lambda
  iter, arm = armijo(x0, opti)

  # Actualizar el valor de x0
  x0 = x0 - arm * grad

  # Hallar el valor de la funcion en x0
  fun = float(opti.subs(zip((x1, x2), x0)))

  # Error
  mod = np.linalg.norm(x0 - (0,0))

  # Introducir el numero de iteraciones, iteraciones internas, lambda, x0 y fun(x0)
  data.append([k, iter, x0, fun, mod])
```

```
  k += 1

# Crear DataFrame de pandas
df = pd.DataFrame(data, columns=['Iteracion', 'Iteraciones Internas', 'x0', 'f(x0)',
     '|| x0 - x||'])

# Imprimir la informacion obtenida
print(df)

# Imprimir el tiempo de ejecucion del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecucion: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-1.2, 1.2, 100)
x2_vals = np.linspace(-1.2, 1.2, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la funcion en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))

# Crear el grafico de las curvas de nivel
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Funcion $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos
     $x_0$')

# Leyenda
plt.legend()

# Mostrar la grafica
plt.show()
```

Listing 3: Código para método de búsqueda de Armijo

Explicación del Código

- `import sympy as sp, numpy as np, pandas as pd, matplotlib.pyplot as plt, time`: Importamos las bibliotecas necesarias para trabajar con funciones simbólicas (`sympy`), manejar datos numéricos (`numpy`), crear gráficos (`matplotlib`) y medir el tiempo de ejecución (`time`).
- `x1, x2 = sp.symbols('x1 x2')`: Definimos las variables simbólicas x_1 y x_2 usando `sympy`.
- `opti = -sp.exp(-((x1 ** 2) + (x2 ** 2)))`: Definimos la función a optimizar, $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$.
- `x0 = np.array([-1.2, 1], dtype=np.float64)`: Establecemos el punto inicial x_0.
- `data = []`: Inicializamos una lista para almacenar los datos de cada iteración.
- `def armijo(x0, opti)`: Definimos la función para realizar la búsqueda de Armijo del paso óptimo λ.
- `while k < 101`: Iniciamos el bucle de iteraciones del método del gradiente.
- `grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1, x2)])`: Calculamos el gradiente de la función en el punto x_0.
- `Normadf = (grad[0]**2 + grad[1]**2)**(0.5)`: Calculamos la norma del gradiente.
- `if Normadf < error:  break`: Condición de terminación del algoritmo.
- `iter, arm = armijo(x0, opti)`: Obtenemos el número de iteraciones internas y el valor de λ mediante búsqueda de Armijo.
- `x0 = x0 - arm * grad`: Actualizamos el valor de x_0.
- `fun = float(opti.subs(zip((x1, x2), x0)))`: Calculamos el valor de la función en x_0.

- `mod = np.linalg.norm(x0 - (0,0))`: Calculamos el error.
- `data.append([k, iter, x0, fun, mod])`: Añadimos los datos de la iteración actual a la lista.
- `df = pd.DataFrame(data, columns=['Iteración', 'Iteraciones Internas', 'x0', 'f(x0)', '|| x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados.
- `print(df)`: Imprimimos la tabla.
- `execution_time = time.time() - start_time`: Calculamos el tiempo de ejecución del programa.
- `plt.contour(X1, X2, Z, levels=20)`: Creamos el gráfico de las curvas de nivel.
- `plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos` x_0`')`: Extraemos y graficamos los puntos x_0 del DataFrame.
- `plt.show()`: Mostramos la gráfica.

Ejercicio 5

$$f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$$

Vamos a aplicar el método de Newton con búsqueda exacta y de Armijo. A continuación, se presenta el código y su explicación.

Método de Newton con Búsqueda de Armijo

```
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

# Definimos el polinomio a optimizar
x1, x2 = sp.symbols('x1 x2')
opti = -sp.exp(-((x1 ** 2) + (x2 ** 2)))

# Variables iniciales
x0 = np.array([0.3, 0.2], dtype=np.float64)
k = 1
error = 1e-3
data1 = []

# Crearemos la funci n armijo
def armijo(x0, opti):
    b, s, o, k1, m = 0.5, 1, 0.1, 1, 0

    while True:
        # Hallamos el valor de lambda
        lmb = s * (b ** m)
        # Hallamos f(x0)
        f = float(opti.subs(zip((x1, x2), x0)))
```

```
        # Hallamos el valor de la gradiente
        grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (
            x1, x2)])
        # Hallamos el valor de f(x0 - lmb * grad)
        f_k = float(opti.subs(zip((x1, x2), x0 - lmb * grad)))
        mod = grad[0] ** 2 + grad[1] ** 2

        # Condici n de parada
        if f_k <= f - o * lmb * mod:
            break
        else:
            m += 1
            k1 += 1

    return k1, lmb

# Iniciar el contador de tiempo
start_time = time.time()

# M todo de Newton
while k < 101:
    # Obtenemos la gradiente de la funci n
    grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1,
        x2)])

    # Calcular la hessiana simb licamente
    hess = sp.hessian(opti, (x1, x2))

    # Sustituir los valores de x0 en la hessiana
    hess_x0 = np.array(hess.subs(zip((x1, x2), x0))).astype(np.float64)

    # Obtenemos la norma de la funci n
    Normadf = (grad[0]**2 + grad[1]**2)**(0.5)

    if Normadf < error:
        break

    # Hallamos el valor de d, mediante la soluci n del sistema lineal
    d = np.linalg.solve(hess_x0, -grad)

    # Obtendremos el n mero de iteraciones internas y el valor de lambda
    iter, arm = armijo(x0, opti)

    # Actualizaremos x0
    x0 = x0 + arm * d
```

```
    # Hallaremos el valor de la funci n en x0
    fun = float(opti.subs(zip((x1, x2), x0)))

    # Error
    mod = np.linalg.norm(x0 - (0,0))

    # Introduciremos el n mero de iteraciones, iteraciones internas, lambda, x0 y
        fun(x0)
    data1.append([k, iter, x0, fun, mod])

    k += 1

# Crear DataFrame de pandas
df = pd.DataFrame(data1, columns=['k', 'Iteraci n interna', 'x0', 'fun(x0)', '|| x0
     - x||'])

# Imprimir la informaci n obtenida
print(df)

# Imprimir el tiempo de ejecuci n del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecuci n: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-0.5, 0.5, 100)
x2_vals = np.linspace(-0.5, 0.5, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la funci n en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))

# Crear el gr fico de las curvas de nivel
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Funci n $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos
     $x_0$')
```

```
# A adir leyenda
plt.legend()

# Mostrar la gr fica
plt.show()
```

Listing 4: Código para método de Newton con búsqueda de Armijo

Explicación del Código

- `import sympy as sp, numpy as np, pandas as pd, matplotlib.pyplot as plt, time`: Importamos las bibliotecas necesarias para trabajar con funciones simbólicas (`sympy`), manejar datos numéricos (`numpy`), crear gráficos (`matplotlib`) y medir el tiempo de ejecución (`time`).
- `x1, x2 = sp.symbols('x1 x2')`: Definimos las variables simbólicas x_1 y x_2 usando `sympy`.
- `opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))`: Definimos la función a optimizar, $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$.
- `x0 = np.array([0.3, 0.2], dtype=np.float64)`: Establecemos el punto inicial x_0.
- `data1 = []`: Inicializamos una lista para almacenar los datos de cada iteración.
- `def armijo(x0, opti)`: Definimos la función para realizar la búsqueda de Armijo del paso óptimo λ.
- `while k < 101`: Iniciamos el bucle de iteraciones del método de Newton.
- `grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1, x2)])`: Calculamos el gradiente de la función en el punto x_0.
- `hess = sp.hessian(opti, (x1, x2))`: Calculamos la hessiana de la función simbólicamente.

- `hess_x0 = np.array(hess.subs(zip((x1, x2), x0))).astype(np.float64)`: Sustituimos los valores de x_0 en la hessiana.

- `Normadf = (grad[0]**2 + grad[1]**2)**(0.5)`: Calculamos la norma del gradiente.

- `if Normadf < error:  break`: Condición de terminación del algoritmo.

- `d = np.linalg.solve(hess_x0, -grad)`: Hallamos el valor de d, mediante la solución del sistema lineal.

- `iter, arm = armijo(x0, opti)`: Obtenemos el número de iteraciones internas y el valor de λ mediante búsqueda de Armijo.

- `x0 = x0 + arm * d`: Actualizamos el valor de x_0.

- `fun = float(opti.subs(zip((x1, x2), x0)))`: Calculamos el valor de la función en x_0.

- `mod = np.linalg.norm(x0 - (0,0))`: Calculamos el error.

- `data1.append([k, iter, x0, fun, mod])`: Añadimos los datos de la iteración actual a la lista.

- `df = pd.DataFrame(data1, columns=['k', 'Iteración interna', 'x0', 'fun(x0)', '|| x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados.

- `print(df)`: Imprimimos la tabla.

- `execution_time = time.time() - start_time`: Calculamos el tiempo de ejecución del programa.

- `plt.contour(X1, X2, Z, levels=20)`: Creamos el gráfico de las curvas de nivel.

- `plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos` x_0`')`: Extraemos y graficamos los puntos x_0 del DataFrame.

- `plt.show()`: Mostramos la gráfica.

Método de Newton con Búsqueda Exacta

```
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
from scipy.optimize import minimize_scalar

# Definimos el polinomio a optimizar
x1, x2 = sp.symbols('x1 x2')
opti = -sp.exp(-((x1 ** 2) + (x2 ** 2)))

# Variables iniciales
x0 = np.array([1, 2], dtype=np.float64)
k = 1
error = 1e-3
data1 = []

# M todo de b squeda unidimensional para encontrar el mejor lambda
def busqueda_unidimensional(x0, d, opti):
    # Definimos la funci n en t rminos de lambda
    lambda_var = sp.symbols('lambda')
    x_new = x0 + lambda_var * d
    f_lambda = opti.subs(zip((x1, x2), x_new))

    # Convertimos f_lambda a una funci n de numpy
    f_lambda_func = sp.lambdify(lambda_var, f_lambda, 'numpy')

    # Usamos minimize_scalar para encontrar el valor  ptimo  de lambda
    res = minimize_scalar(f_lambda_func)

    return res.x

# Iniciar el contador de tiempo
start_time = time.time()

# M todo de Newton
while k < 101:
    # Obtenemos la gradiente de la funci n
    grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1,
        x2)])
```

```python
    # Calcular la hessiana simb licamente
    hess = sp.hessian(opti, (x1, x2))

    # Sustituir los valores de x0 en la hessiana
    hess_x0 = np.array(hess.subs(zip((x1, x2), x0))).astype(np.float64)

    # Obtenemos la norma de la funci n
    Normadf = (grad[0]**2 + grad[1]**2)**(0.5)

    if Normadf < error:
        break

    # Hallamos el valor de d, mediante la soluci n del sistema lineal
    d = np.linalg.solve(hess_x0, -grad)

    # Obtendremos el valor de lambda usando la b squeda unidimensional
    arm = busqueda_unidimensional(x0, d, opti)

    # Actualizaremos x0
    x0 = x0 + arm * d

    # Hallaremos el valor de la funci n en x0
    fun = float(opti.subs(zip((x1, x2), x0)))

    # Error
    mod = np.linalg.norm(x0 - (0, 0))

    # Introduciremos el n mero de iteraciones, lambda, x0 y fun(x0)
    data1.append([k, 1, x0, fun, mod])

    k += 1

# Crear DataFrame de pandas
df = pd.DataFrame(data1, columns=['k', 'Iteraci n interna', 'x0', 'fun(x0)', '|| x0
    - x||'])

# Imprimir la informaci n obtenida
print(df)

# Imprimir el tiempo de ejecuci n del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecuci n: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-0.5, 0.5, 100)
```

```
x2_vals = np.linspace(-0.5, 0.5, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la funci n en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))

# Crear el gr fico de las curvas de nivel
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Funci n $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos
    $x_0$')

# A adir leyenda
plt.legend()

# Mostrar la gr fica
plt.show()
```

Listing 5: Código para método de Newton con búsqueda exacta

Explicación del Código

- `import sympy as sp, numpy as np, pandas as pd, matplotlib.pyplot as plt, time, from scipy.optimize import minimize_scalar`: Importamos las bibliotecas necesarias para trabajar con funciones simbólicas (`sympy`), manejar datos numéricos (`numpy`), crear gráficos (`matplotlib`), medir el tiempo de ejecución (`time`) y realizar optimización escalar (`minimize_scalar`).
- `x1, x2 = sp.symbols('x1 x2')`: Definimos las variables simbólicas x_1 y x_2 usando `sympy`.
- `opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))`: Definimos la función a optimizar, $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$.

- `x0 = np.array([1, 2], dtype=np.float64)`: Establecemos el punto inicial x_0.
- `data1 = []`: Inicializamos una lista para almacenar los datos de cada iteración.
- `def busqueda_unidimensional(x0, d, opti)`: Definimos la función para realizar la búsqueda unidimensional del paso óptimo λ.
- `while k < 101`: Iniciamos el bucle de iteraciones del método de Newton.
- `grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1, x2)])`: Calculamos el gradiente de la función en el punto x_0.
- `hess = sp.hessian(opti, (x1, x2))`: Calculamos la hessiana de la función simbólicamente.
- `hess_x0 = np.array(hess.subs(zip((x1, x2), x0))).astype(np.float64)`: Sustituimos los valores de x_0 en la hessiana.
- `Normadf = (grad[0]**2 + grad[1]**2)**(0.5)`: Calculamos la norma del gradiente.
- `if Normadf < error:  break`: Condición de terminación del algoritmo.
- `d = np.linalg.solve(hess_x0, -grad)`: Hallamos el valor de d, mediante la solución del sistema lineal.
- `arm = busqueda_unidimensional(x0, d, opti)`: Obtenemos el valor de λ usando la búsqueda unidimensional.
- `x0 = x0 + arm * d`: Actualizamos el valor de x_0.
- `fun = float(opti.subs(zip((x1, x2), x0)))`: Calculamos el valor de la función en x_0.
- `mod = np.linalg.norm(x0 - (0, 0))`: Calculamos el error.
- `data1.append([k, 1, x0, fun, mod])`: Añadimos los datos de la iteración actual a la lista.

- `df = pd.DataFrame(data1, columns=['k', 'Iteración interna', 'x0', 'fun(x0)', '|| x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados.
- `print(df)`: Imprimimos la tabla.
- `execution_time = time.time() - start_time`: Calculamos el tiempo de ejecución del programa.
- `plt.contour(X1, X2, Z, levels=20)`: Creamos el gráfico de las curvas de nivel.
- `plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Puntos` x_0`')`: Extraemos y graficamos los puntos x_0 del DataFrame.
- `plt.show()`: Mostramos la gráfica.

Ejercicio 6

$$f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$$

Vamos a aplicar el método de Cuasi-Newton con búsqueda exacta y de Armijo. A continuación, se presenta el código y su explicación.

```
import numpy as np
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import sympy as sp

# Definir la funci n y su gradiente
def f(x):
    return -np.exp(-(x[0]**2 + x[1]**2))

def grad_f(x):
    exp_term = np.exp(-(x[0]**2 + x[1]**2))
    return np.array([2 * x[0] * exp_term, 2 * x[1] * exp_term])

# B squeda exacta usando scipy.optimize
def exact_line_search(x, d):
    phi = lambda alpha: f(x + alpha * d)
    result = minimize_scalar(phi)
    return result.x

# M todo del Cuasi-Newton con B squeda Exacta (BFGS)
def quasi_newton_bfgs_exact(x0, tol=1e-6, max_iter=1000):
    x = x0
    n = len(x)
    H = np.eye(n)  # Inicializar H como la identidad
    iter_count = 0
    data = []

    for _ in range(max_iter):
```

```
        grad = grad_f(x)
        if np.linalg.norm(grad) < tol:
            break

        d = -H @ grad
        alpha = exact_line_search(x, d)
        x_new = x + alpha * d

        s = x_new - x
        y = grad_f(x_new) - grad

        if np.dot(y, s) > 0:  # Para asegurar que la actualizaci n sea positiva
            rho = 1.0 / np.dot(y, s)
            I = np.eye(n)
            H = (I - rho * np.outer(s, y)) @ H @ (I - rho * np.outer(y, s)) + rho *
                np.outer(s, s)

        x = x_new
        data.append([iter_count, 1, x, f(x), np.linalg.norm(x)])
        iter_count += 1

    return x, iter_count, data

# M todo del Cuasi-Newton con el Criterio de Armijo (BFGS)
def quasi_newton_bfgs_armijo(x0, tol=1e-6, max_iter=1000, alpha=1, beta=0.5, sigma=1
    e-4):
    x = x0
    n = len(x)
    H = np.eye(n)  # Inicializar H como la identidad
    iter_count = 0
    total_internal_iter_count = 0
    data = []

    for _ in range(max_iter):
        grad = grad_f(x)
        if np.linalg.norm(grad) < tol:
            break

        d = -H @ grad
        t = alpha
        internal_iter_count = 0

        while f(x + t * d) > f(x) + sigma * t * np.dot(grad, d):
            t *= beta
            internal_iter_count += 1
```

```
        x_new = x + t * d
        s = x_new - x
        y = grad_f(x_new) - grad

        if np.dot(y, s) > 0:  # Para asegurar que la actualizaci n sea positiva
            rho = 1.0 / np.dot(y, s)
            I = np.eye(n)
            H = (I - rho * np.outer(s, y)) @ H @ (I - rho * np.outer(y, s)) + rho *
                np.outer(s, s)

        x = x_new
        data.append([iter_count, internal_iter_count, x, f(x), np.linalg.norm(x)])
        iter_count += 1
        total_internal_iter_count += internal_iter_count

    return x, iter_count, total_internal_iter_count, data

# Inicializaci n y llamada al m todo con b squeda exacta
x0 = np.array([1.0, 1.0])
result_exact, iter_count_exact, data_exact = quasi_newton_bfgs_exact(x0)
print("Resultado (Cuasi-Newton con B squeda Exacta):", result_exact)
print(" N mero de iteraciones:", iter_count_exact)

# Inicializaci n y llamada al m todo con Armijo
result_armijo, iter_count_armijo, total_internal_iter_count_armijo, data_armijo =
    quasi_newton_bfgs_armijo(x0)
print("Resultado (Cuasi-Newton con Armijo):", result_armijo)
print(" N mero de iteraciones externas:", iter_count_armijo)
print(" N mero de iteraciones internas:", total_internal_iter_count_armijo)

# Crear un DataFrame de pandas para ambos m todos
import pandas as pd

df_exact = pd.DataFrame(data_exact, columns=['Iteraci n', 'Iteraciones internas', '
    x0', 'f(x0)', '||x0 - x||'])
df_armijo = pd.DataFrame(data_armijo, columns=['Iteraci n', 'Iteraciones internas',
     'x0', 'f(x0)', '||x0 - x||'])

# Imprimir los DataFrames
print(" M todo de B squeda Exacta:")
print(df_exact)

print(" M todo de Armijo:")
print(df_armijo)
```

```
# Graficar las curvas de nivel
x1_vals = np.linspace(-2, 2, 400)
x2_vals = np.linspace(-2, 2, 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)
F = -np.exp(-(X1**2 + X2**2))

plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, F, levels=np.linspace(-1, 0, 10), cmap='viridis')
plt.colorbar(contour)
plt.title('Curvas de Nivel de $f(x_1, x_2) = -e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Graficar los puntos para ambos m todos
x0_points_exact = np.array([point[2] for point in data_exact])
x0_points_armijo = np.array([point[2] for point in data_armijo])
plt.scatter(x0_points_exact[:, 0], x0_points_exact[:, 1], color='blue', marker='x',
    label='Exacta')
plt.scatter(x0_points_armijo[:, 0], x0_points_armijo[:, 1], color='red', marker='o',
     label='Armijo')

plt.legend()
plt.show()
```

Listing 6: Código para método de Cuasi-Newton con búsqueda exacta y de Armijo

Explicación del Código

- `import numpy as np, scipy.optimize import minimize_scalar, matplotlib.pyplot as plt, sympy as sp`: Importamos las bibliotecas necesarias para trabajar con funciones, optimización y gráficos.
- `def f(x)`: Definimos la función a optimizar, $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$.
- `def grad_f(x)`: Definimos el gradiente de la función.
- `def exact_line_search(x, d)`: Definimos la función para realizar la búsqueda exacta del paso óptimo α.
- `def quasi_newton_bfgs_exact(x0, tol=1e-6, max_iter=1000)`: Definimos el método de Cuasi-Newton con búsqueda exacta.

- `def quasi_newton_bfgs_armijo(x0, tol=1e-6, max_iter=1000, alpha=1, beta=0.5, sigma=1e-4)`: Definimos el método de Cuasi-Newton con búsqueda de Armijo.
- `x0 = np.array([1.0, 1.0])`: Establecemos el punto inicial x_0.
- `result_exact, iter_count_exact, data_exact = quasi_newton_bfgs_exact(x0)`: Ejecutamos el método de Cuasi-Newton con búsqueda exacta y almacenamos los resultados.
- `result_armijo, iter_count_armijo, total_internal_iter_count_armijo, data_armijo = quasi_newton_bfgs_armijo(x0)`: Ejecutamos el método de Cuasi-Newton con búsqueda de Armijo y almacenamos los resultados.
- `df_exact = pd.DataFrame(data_exact, columns=['Iteración', 'Iteraciones internas', 'x0', 'f(x0)', '||x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados del método de búsqueda exacta.
- `df_armijo = pd.DataFrame(data_armijo, columns=['Iteración', 'Iteraciones internas', 'x0', 'f(x0)', '||x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados del método de búsqueda de Armijo.
- `plt.contour(X1, X2, F, levels=np.linspace(-1, 0, 10), cmap='viridis')`: Creamos el gráfico de las curvas de nivel.
- `plt.scatter(x0_points_exact[:, 0], x0_points_exact[:, 1], color='blue', marker='x', label='Exacta')`: Graficamos los puntos del método de búsqueda exacta.
- `plt.scatter(x0_points_armijo[:, 0], x0_points_armijo[:, 1], color='red', marker='o', label='Armijo')`: Graficamos los puntos del método de búsqueda de Armijo.
- `plt.show()`: Mostramos la gráfica.

Ejercicio 7

$$f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$$

Vamos a aplicar el método del gradiente conjugado con búsqueda unidimensional y de Armijo. A continuación, se presenta el código y su explicación.

```
import numpy as np
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import sympy as sp

# Definir la funci n y su gradiente
def f(x):
    return -np.exp(-(x[0]**2 + x[1]**2))

def grad_f(x):
    exp_term = np.exp(-(x[0]**2 + x[1]**2))
    return np.array([2 * x[0] * exp_term, 2 * x[1] * exp_term])

# B squeda exacta usando scipy.optimize
def exact_line_search(x, d):
    phi = lambda alpha: f(x + alpha * d)
    result = minimize_scalar(phi)
    return result.x

# M todo del Gradiente Conjugado con B squeda Exacta
def conjugate_gradient_exact(x0, tol=1e-6, max_iter=1000):
    x = x0
    grad = grad_f(x)
    d = -grad
    iter_count = 0
    data = []

    for _ in range(max_iter):
```

```
        if np.linalg.norm(grad) < tol:
            break
        alpha = exact_line_search(x, d)
        x_new = x + alpha * d
        grad_new = grad_f(x_new)
        beta = np.dot(grad_new, grad_new) / np.dot(grad, grad)
        d = -grad_new + beta * d
        x = x_new
        grad = grad_new
        iter_count += 1
        data.append([iter_count, 1, x, f(x), np.linalg.norm(x)])

    return x, iter_count, data

# M todo del Gradiente Conjugado con el Criterio de Armijo
def conjugate_gradient_armijo(x0, tol=1e-6, max_iter=1000, alpha=1, beta=0.5, sigma
    =1e-4):
    x = x0
    grad = grad_f(x)
    d = -grad
    iter_count = 0
    total_internal_iter_count = 0
    data = []

    for _ in range(max_iter):
        if np.linalg.norm(grad) < tol:
            break
        t = alpha
        internal_iter_count = 0
        while f(x + t * d) > f(x) + sigma * t * np.dot(grad, d):
            t *= beta
            internal_iter_count += 1

        x_new = x + t * d
        grad_new = grad_f(x_new)
        beta_cg = np.dot(grad_new, grad_new) / np.dot(grad, grad)
        d = -grad_new + beta_cg * d
        x = x_new
        grad = grad_new
        iter_count += 1
        total_internal_iter_count += internal_iter_count
        data.append([iter_count, internal_iter_count, x, f(x), np.linalg.norm(x)])

    return x, iter_count, total_internal_iter_count, data
```

```
# Inicializaci n y llamada al m todo con b squeda exacta
x0 = np.array([1.0, 1.0])
result_exact, iter_count_exact, data_exact = conjugate_gradient_exact(x0)
print("Resultado (Gradiente Conjugado con B squeda Exacta):", result_exact)
print(" N mero de iteraciones:", iter_count_exact)

# Inicializaci n y llamada al m todo con Armijo
result_armijo, iter_count_armijo, total_internal_iter_count_armijo, data_armijo =
    conjugate_gradient_armijo(x0)
print("Resultado (Gradiente Conjugado con Armijo):", result_armijo)
print(" N mero de iteraciones externas:", iter_count_armijo)
print(" N mero de iteraciones internas:", total_internal_iter_count_armijo)

# Crear un DataFrame de pandas para ambos m todos
import pandas as pd

df_exact = pd.DataFrame(data_exact, columns=['Iteraci n', 'Iteraciones internas', '
    x0', 'f(x0)', '||x0 - x||'])
df_armijo = pd.DataFrame(data_armijo, columns=['Iteraci n', 'Iteraciones internas',
     'x0', 'f(x0)', '||x0 - x||'])

# Imprimir los DataFrames
print(" M todo de B squeda Exacta:")
print(df_exact)

print(" M todo de Armijo:")
print(df_armijo)

# Graficar las curvas de nivel
x1_vals = np.linspace(-2, 2, 400)
x2_vals = np.linspace(-2, 2, 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)
F = -np.exp(-(X1**2 + X2**2))

plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, F, levels=np.linspace(-1, 0, 10), cmap='viridis')
plt.colorbar(contour)
plt.title('Curvas de Nivel de $f(x_1, x_2) = -e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Graficar los puntos para ambos m todos
x0_points_exact = np.array([point[2] for point in data_exact])
x0_points_armijo = np.array([point[2] for point in data_armijo])
```

```
plt.scatter(x0_points_exact[:, 0], x0_points_exact[:, 1], color='blue', marker='x',
    label='Exacta')
plt.scatter(x0_points_armijo[:, 0], x0_points_armijo[:, 1], color='red', marker='o',
     label='Armijo')

plt.legend()
plt.show()
```

Listing 7: Código para método del gradiente conjugado con búsqueda unidimensional y de Armijo

Explicación del Código

- `import numpy as np, scipy.optimize import minimize_scalar, matplotlib.pyplot as plt, sympy as sp`: Importamos las bibliotecas necesarias para trabajar con funciones, optimización y gráficos.
- `def f(x)`: Definimos la función a optimizar, $f(x_1, x_2) = -e^{-(x_1^2+x_2^2)}$.
- `def grad_f(x)`: Definimos el gradiente de la función.
- `def exact_line_search(x, d)`: Definimos la función para realizar la búsqueda exacta del paso óptimo α.
- `def conjugate_gradient_exact(x0, tol=1e-6, max_iter=1000)`: Definimos el método del gradiente conjugado con búsqueda exacta.
- `def conjugate_gradient_armijo(x0, tol=1e-6, max_iter=1000, alpha=1, beta=0.5, sigma=1e-4)`: Definimos el método del gradiente conjugado con búsqueda de Armijo.
- `x0 = np.array([1.0, 1.0])`: Establecemos el punto inicial x_0.
- `result_exact, iter_count_exact, data_exact = conjugate_gradient_exact(x0)`: Ejecutamos el método del gradiente conjugado con búsqueda exacta y almacenamos los resultados.
- `result_armijo, iter_count_armijo, total_internal_iter_count_armijo,`

`data_armijo = conjugate_gradient_armijo(x0)`: Ejecutamos el método del gradiente conjugado con búsqueda de Armijo y almacenamos los resultados.

- `df_exact = pd.DataFrame(data_exact, columns=['Iteración', 'Iteraciones internas', 'x0', 'f(x0)', '||x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados del método de búsqueda exacta.

- `df_armijo = pd.DataFrame(data_armijo, columns=['Iteración', 'Iteraciones internas', 'x0', 'f(x0)', '||x0 - x||'])`: Creamos un DataFrame de pandas con los datos recolectados del método de búsqueda de Armijo.

- `plt.contour(X1, X2, F, levels=np.linspace(-1, 0, 10), cmap='viridis')`: Creamos el gráfico de las curvas de nivel.

- `plt.scatter(x0_points_exact[:, 0], x0_points_exact[:, 1], color='blue', marker='x', label='Exacta')`: Graficamos los puntos del método de búsqueda exacta.

- `plt.scatter(x0_points_armijo[:, 0], x0_points_armijo[:, 1], color='red', marker='o', label='Armijo')`: Graficamos los puntos del método de búsqueda de Armijo.

- `plt.show()`: Mostramos la gráfica.

Ejercicio 9

En esta sección se comparan los distintos métodos aplicados en términos de iteraciones externas, iteraciones internas y tiempo de ejecución.

Método	**Externas**	**Internas(Máx)**	**Tiempo (s)**
Gradiente con Búsqueda Exacta	1	1	0.265
Gradiente con Criterio de Armijo	5	2	0.050
Newton Puro con Búsqueda Exacta	1	1	0.022
Newton Puro con Criterio de Armijo	9	2	0.182
Cuasi-Newton con Búsqueda Exacta	1	1	0.050
Cuasi-Newton con Criterio de Armijo	6	4	0.120
Gradiente Conj. con Búsqueda Exacta	1	1	0.030
Gradiente Conj. con Criterio de Armijo	16	2	0.150

Table 1: Cuadro Comparativo de Métodos de Optimización

Análisis y Conclusiones

De los resultados obtenidos se concluye que:

- **Método del Gradiente con Búsqueda Exacta**: Es eficiente y rápido en convergencia, adecuado para funciones suaves y doblemente diferenciables.
- **Método del Gradiente con Criterio de Armijo**: Es mas robusta en comparación con la búsqueda exacta, pero requiere más iteraciones internas.
- **Newton Puro con Búsqueda Exacta**: Es el más rápido y eficiente en

términos de iteraciones y tiempo, especialmente adecuado para funciones con derivadas de segundo orden.

- **Newton Puro con Criterio de Armijo**: Aunque también es robusto, requiere más iteraciones y tiempo en comparación con la búsqueda exacta.
- **Cuasi-Newton con Búsqueda Exacta y Criterio de Armijo**: Ofrecen un balance entre eficiencia y robustez. El método con búsqueda exacta es más rápido pero menos robusto.
- **Gradiente Conjugado con Búsqueda Exacta y Criterio de Armijo**: Muestran buen desempeño, siendo el método con Armijo más robusto pero más lento.

En resumen, el **método de Newton Puro con Búsqueda Exacta** demostró ser el más eficiente en este ejercicio específico debido a su rápida convergencia y bajo tiempo de ejecución. Si bien parece ser el más óptimo, solo lo fue en este ejercicio por ser doblemente diferénciable, caso contrario no seria tan eficiente.