402 MÉTODO DE DESCENSO DE GRADIENTE

September 25, 2025

1 Introducción

El **método de descenso de gradiente** es un algoritmo iterativo diseñado para encontrar el mínimo de una función diferenciable $f(\mathbf{x})$, donde $\mathbf{x} \in \mathbb{R}^n$. Su principio fundamental consiste en moverse, desde un punto inicial, en la dirección que garantiza la disminución más pronunciada del valor de la función.

La idea central es que para cualquier punto $\mathbf{x}$ en el dominio de la función, el gradiente $\nabla f(\mathbf{x})$ apunta en la dirección de máximo crecimiento. En consencuencia, la dirección opuesta, $-\nabla f(\mathbf{x})$, representa la dirección de **máximo descenso**. El algoritmo utiliza esta propiedad para construir una secuencia de puntos $\{\mathbf{x}_k\}$ que converge hacia un mínimo de la función.

El proceso comienza con una estimación inicial $\mathbf{x}_0$ y, a partir de ella, genera las siguientes aproximaciones mediante la regla de actualización:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)$$

En esta ecuación, el parámetro $\alpha > 0$, conocido como **tasa de aprendizaje (learning rate)**, es un escalar que determina la longitud del paso en cada iteración. De esta manera, cada nuevo punto $\mathbf{x}_{k+1}$ se obtiene desplazando el punto actual $\mathbf{x}_k$ una distancia α en la dirección opuesta al gradiente.

2 Algoritmo de descenso de gradiente

El procedimiento iterativo del algoritmo se puede resumir en los siguientes pasos:

1. **Inicialización:** Se elige un punto de partida arbitrario $\mathbf{x}_0$ como la primera estimación de la solución.
2. **Cálculo del gradiente:** En la iteración k, se evalúa el gradiente de la función en el punto actual, $\nabla f(\mathbf{x}_k)$.
3. **Actualización:** Se calcula la siguiente aproximación $\mathbf{x}_{k+1}$ moviendose en la dirección opuesta al gradiente, aplicando la regla:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)$$

4. **Verificación de convergencia:** Se comprueba si se ha alcanzado un **criterio de detención** predefinido. Si se cumple, al algoritmo finaliza y retorna $\mathbf{x}_{k+1}$ como solución aproximada. En caso contrario, se vuelve al paso 2 para la siguiente iteración.

3 Criterios de convergencia

Para determinar cuándo deterner el algoritmo, se utilizan criterios que indican que la solución se ha aproximado suficientemente a un mínimo. Las más comunes son:

1. **Magnitud del gradiente:** La norma (o magnitud) del gradiente es carcada a cero. Esto indica que la función se ha "aplanado" y se está cerca de un punto estacionario.

$$\|\nabla f(\mathbf{x}_{k+1})\| < \epsilon$$

2. **Cambio en la posición:** La distancia entre dos iteraciones consecutivas es insignificante, lo que sugiere que el algoritmo ya no está realizando cambios sustanciales en la solución.

$$\|\mathbf{x}_{k+1} - \mathbf{x}_k\| < \epsilon$$

En ambos casos, ϵ es una tolerancia pequeña predefinida (por ejemplo, 10^{-6}).

3.1 Criterio de magnitud del gradiente

El criterio de la **magnitud del gradiente** evalúa si la función es prácticamente "plana" en la iteración actual. Su justificación teórica se basa en la condición de optimalidad de primer orden, la cual establece que el gradiente de una función deben ser el vector nulo, $\nabla f(\mathbf{x}) = 0$, en un punto de mínimo local.

Este criterio responde a la pregunta: *¿Hemos alcanzado un punto donde la pendiente en todas las direcciones es lo suficientemente cercada a cero?.*

Ventajas:

- **Teóricamente correcto:** Es una implementación directa de la condición necesaria para un punto crítico (mínimo, máximo o punto de silla). Si la norma del gradiente es cercana a cero, garantizamos que estamos próximos a uno de estos puntos.
- **Confiable en valles planos:** En funciones que presentan regiones extensas y con poca curvatura (valles planos), el cambio entre iteraciones, $\mathbf{x}_{k+1} - \mathbf{x}_k$, puede volverse muy pequeño, deteniendo prematuramente el algoritmo. Sin embargo, la magnitud del gradiente, aunque reducida, puede seguir indicando que aún no se ha llegado al verdadero mínimo. Este criterio gestiona estas situaciones con mayor eficacia.

Desventajas:

- **Sensibilidad a la escala:** La magnitud del gradiente depende directamente de la escala de la función objetivo, $f(\mathbf{x})$. Una misma tolerancia, como $\epsilon = 10^{-6}$, puede ser demasiado estricta para una función con gradiente naturalmente pequeños o, por el contrario, demasiado laxa para otra con gradientes muy grandes. Esto puede requerir un ajuste manual de ϵ según el problema.

3.2 Criterio de cambio de posición

El criterio de **cambio de posición**, $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| < \epsilon$, mide la distancia euclidiana entre dos puntos consecutivos del algoritmo. Su objetivo es determinar si las actualizaciones posteriores están generando un cambio significativo en la solución.

Este criterio responde a la pregunta: *¿Se han vuelto los pasos tan pequeños que el progreso hacia el mínimo es despreciable?*.

Ventajas:

- **Intuitivo y sencillo:** Es un concepto fácil de interpretar. Si la posición de la solución deja de cambiar sustancialmente, es razonable suponer que se ha alcanzado una convergencia.
- **Eficaz en funciones convexas simples:** Para problemas con una estructura convexa y suave, donde el mínimo está bien definido y no existen mesetas, este criterio suele ser un indicador confiable de convergencia.

Desventajas:

- **Riesgo de detención prematura:** Su principal debilidad. Si el algoritmo encuentra una región muy plana (una meseta o un valle con pendiente suave), o si la tasa de aprendizaje α es demasiado pequeña, la magnitud de los pasos disminuirá drásticamente. El criterio puede interpretar esta falta de movimiento como una convergencia, cuando en realidad el algoritmo simplemente avanza con lentitud hacia el mínimo.
- **Sensibilidad a la tasa de aprendizaje:** Este criterio está fuertemente acoplado al valor de α. Una tasa de aprendizaje muy pequeña siempre generará pasos cortos, lo que puede provocar una terminación anticipada y engañosa, independientemente de cuán lejos se esté del verdadero punto óptimo.

3.3 Combinación de criterios

En aplicaciones prácticas, un optimizador robusto no debe depender de un único criterio de detención. La estrategia estándar es implementar una combinación de condiciones, donde el algoritmo finaliza en cuanto se satisface la primera de ellas. Este enfoque híbrido proporciona un equilibrio entre la justificación teórica y la eficiencia computacional.

Las condiciones más utilizadas son:

- **Magnitud del gradiente:** Asegura que la solución se encuentra en la proximidad de un punto estacionario, cumpliendo con la condición de optimalidad.
- **Cambio en la posición:** Actúa como un detector de estancamiento, deteniendo el algoritmo si el progreso entre iteraciones se vuelve insignificante.
- **Número máximo de iteraciones:** Es un mecanismo de seguridad esencial. Garantiza que el algoritmo terminará en un tiempo finito, evitando bucles infinitos en casos de divergencia o convergencia extremadamente lenta.

Si bien la **magnitud del gradiente** es considerado el criterio más fiable desde una perspectiva teórica, la implementación profesional y más segura consiste en utilizar los tres criterios de manera conjunta. Esto asegura que el algoritmo no solo se detenga por las razones correctas (convergencia a un mínimo o estancamiento), sino que también cuente con una salvaguarda que previene ejecuciones indefinidas.

4 Elección de la tasa de aprendizaje

La selección de una tasa de aprendizaje (α) adecuada es uno de los aspectos más críticos en la implementación del descenso de gradiente, ya que determina la estabilidad y la velocidad de convergencia del algoritmo.

- **Tasa de aprendizaje alta:** Un valor alto puede acelerar la convergencia, ya que permite dar pasos más grandes hacia el mínimo. Sin embargo, si es demasiado grande, corre el riesgo de "saltarse" el mínimo, oscilando alrededor de él o, en el peor de los casos, divergiendo por completo.
- **Tasa de aprendizaje baja:** Un valor pequeño garantiza que el algoritmo no se salte el mínimo, produciendo un descenso estable y seguro. La desventaja es que la convergencia puede ser extremadamente lenta, requiriendo un número mucho mayor de iteraciones y, por lo tanto, más tiempo de cómputo.

El desafío consiste en encontrar un equilibrio: un valor de α lo suficientemente grande para converger rápidamente, pero lo suficientemente pequeño para no diverger.

5 Implementación

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

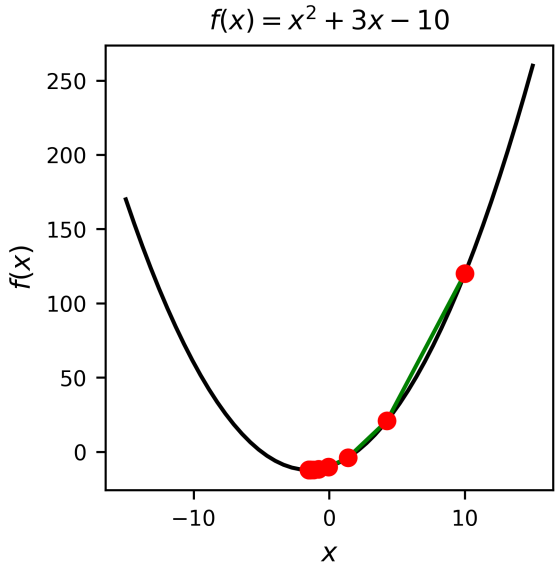
[2]:
```
resolucion = 600
```

[3]:
```
# ----------------------------------
# Encontrar el mínimo de la función:
# f(x) = x ** 2 + 3 * x - 10
# ----------------------------------

# Definición de la función objetivo y su gradiente
def f_uni(x):
    return x ** 2 + 3 * x - 10

def grad_f_uni(x):
    return 2 * x + 3

# Algoritmo de descenso de gradiente
def dg_uni(x_actual, alpha, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        x_nuevo = x_actual - alpha * grad_f_uni(x_actual)
        x_historial = np.append(x_historial, x_nuevo)
        criterio_1 = np.linalg.norm(grad_f_uni(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
```

```
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial

# Ajustes de parámetros del algoritmo
alpha = 0.25

# Definición del punto inicial
x_actual = 10

# Ejecución del algoritmo y resultados
r_uni = dg_uni (x_actual, alpha)

print("x =", r_uni[0])
print("f(x) =", f_uni(r_uni[0]))
print("Iteraciones =", len(r_uni[1]))
```

```
x = -1.499999314546585
f(x) = -12.249999999999531
Iteraciones = 25
```

[4]:
```
# Graficar la función
x = np.linspace(-15, 15, 50)
y = f_uni(x)

plt.figure(figsize = (3, 3), dpi = resolucion)
plt.plot(x, y, color = "black", zorder = 1)

plt.title("$f(x) = x^2 + 3 x - 10$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$f(x)$", fontsize = 10)

# Puntos de trayectoria
plt.plot(r_uni[1], f_uni(r_uni[1]), color = "green", zorder = 2)
plt.scatter(r_uni[1], f_uni(r_uni[1]), color = "red", zorder = 3)
plt.tick_params(labelsize = 8)
plt.show()
```



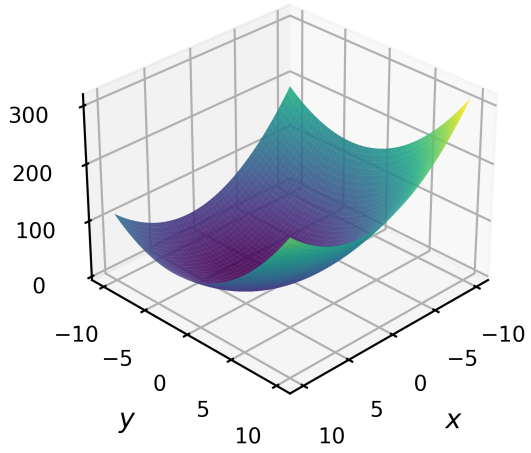
[5]:
```python
# ------------------------------------
# Encontrar el mínimo de la función:
# f(x, y) = (x - 2) ** 2 + (y + 3) ** 2
# ------------------------------------

# Definimos la función objetivo y su gradiente
def f_multi(x):
    return (x[0] - 2) ** 2 + (x[1] + 3) ** 2

def grad_f_multi(x):
    dx = 2 * (x[0] - 2)
    dy = 2 * (x[1] + 3)
    return np.array([dx, dy])

# Algoritmo de descenso de gradiente
def dg_multi(x_actual, alpha, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        x_nuevo = x_actual - alpha * grad_f_multi(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_f_multi(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial
```

```
# Ajuste de parámetros del algoritmo
alpha = 0.1

# Definición del punto inicial
x_actual = np.array([-5, 5])

# Ejecución del algoritmo y resultados
r_multi = dg_multi(x_actual, alpha)

print("(x, y) =", r_multi[0])
print("f(x, y) =", f_multi(r_multi[0]))
print("Iteraciones =", len(r_multi[1]))
```

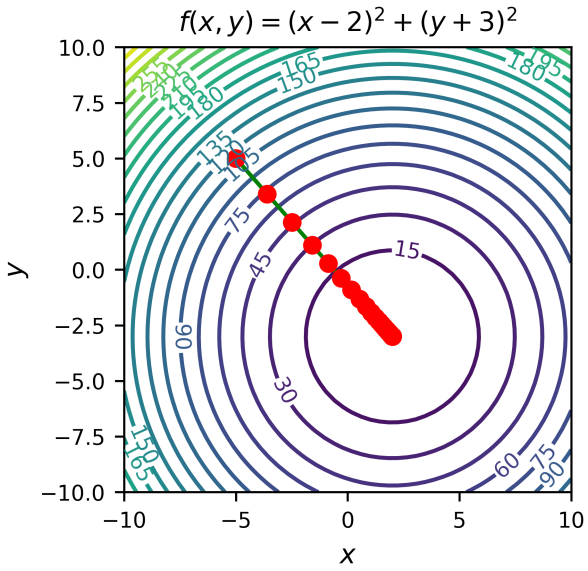
```
(x, y) = [ 1.99999775 -2.99999743]
f(x, y) = 1.1671769449993414e-11
Iteraciones = 68
```

[6]:
```
# Graficar la función
x = np.linspace(-10, 10, 50)
y = np.linspace(-10, 10, 50)
X, Y = np.meshgrid(x, y)
Z = f_multi([X, Y])

fig = plt.figure(figsize = (3, 3), dpi = resolucion)
ax = fig.add_subplot(111, projection = "3d")
ax.plot_surface(X, Y, Z, cmap = "viridis", alpha = 0.9)
ax.view_init(elev = 30, azim = 45)

ax.set_title("$f(x, y) = (x - 2)^2 + (y + 3)^2$", fontsize = 10)
ax.set_xlabel("$x$", fontsize = 10)
ax.set_ylabel("$y$", fontsize = 10)
ax.tick_params(labelsize = 8)
plt.show()
```

$$f(x, y) = (x - 2)^2 + (y + 3)^2$$



[7]:
```
# Graficar la función
x = np.linspace(-10, 10, 50)
y = np.linspace(-10, 10, 50)
X, Y = np.meshgrid(x, y)
Z = f_multi([X, Y])

plt.figure(figsize=(3, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")

plt.clabel(contour, inline = 1, fontsize = 8)
plt.title("$f(x, y) = (x - 2)^2 + (y + 3)^2$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$y$", fontsize = 10)

# Puntos de trayectoria
plt.plot(r_multi[1][:, 0], r_multi[1][:, 1], color = "green", zorder = 2)
plt.scatter(r_multi[1][:, 0], r_multi[1][:, 1], color = "red", zorder = 3)
plt.tick_params(labelsize = 8)
plt.show()
```



[8]:
```python
# ----------------------------------
# Encontrar el mínimo de la función:
# f(x) = (x - 2) ** 2
# ----------------------------------

# Definimos la función objetivo y su gradiente
def f_ejemplo(x):
    return (x - 2) ** 2

def grad_f_ejemplo(x):
    return 2 * (x - 2)

# Algoritmo de descenso de gradiente
def dg_ejemplo(x_inicial, alpha, max_iter = 10000, tol = 1e-6):
    resultados = np.zeros((len(alpha), 3))
    for k in range(len(alpha)):
        x_actual = x_inicial
        for i in range(max_iter):
            x_nuevo = x_actual - alpha[k] * grad_f_ejemplo(x_actual)
            criterio_1 = np.linalg.norm(grad_f_ejemplo(x_nuevo))
            criterio_2 = np.linalg.norm(x_nuevo - x_actual)
            if (criterio_1 < tol or criterio_2 < tol):
                resultados[k, 0] = alpha[k]
                resultados[k, 1] = x_nuevo
                resultados[k, 2] = i
                break
            x_actual = x_nuevo
```

```
        df_resultados = pd.DataFrame(resultados, columns = ["Alpha", "Mínimo",␣
↪"Iteraciones"])
    return df_resultados

# Parámetros del algoritmo
x_inicial = 10

# Vector con valores de alpha
alpha = np.array([0.05, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90])

# Ejecución del algoritmo y resultados
r_ejemplo = dg_ejemplo(x_inicial, alpha)

print(r_ejemplo)
```

```
   Alpha    Mínimo  Iteraciones
0   0.05  2.000008        130.0
1   0.10  2.000003         65.0
2   0.20  2.000001         30.0
3   0.30  2.000001         17.0
4   0.40  2.000000         10.0
5   0.50  2.000000          0.0
6   0.60  2.000000         10.0
7   0.70  2.000000         18.0
8   0.80  2.000000         32.0
9   0.90  2.000000         74.0
```

6 Decaimiento de la tasa de aprendizaje

Para mejorar la eficiencia y estabilidad del descenso de gradiente, existen métodos que ajustan la tasa de aprendizaje automáticamente durante el proceso de optimización. Uno de los enfoques más comunes es el **decaimiento de la tasa de aprendizaje** (learning rate decay).

La idea principal es comenzar con una tasa de aprendizaje relativamente alta y reducirla gradualmente a medida que avanzan las iteraciones. Esto permite:

- **Pasos grandes al inicio:** Cuando el algoritmo está lejos del mínimo, se avanza rápidamente hacia la región de la solución.
- **Pasos pequeños al final:** A medida que el algoritmo se acerca al mínimo, los pasos se reducen para realizar un ajuste fino y evitar "saltarse" la solución óptima.

Una fórmula común para implementar este decaimiento es:

$$\alpha_k = \frac{\alpha_0}{1 + k\tau}$$

En esta ecuación, α_k es la tasa de aprendizaje en la iteración k, α_0 es la tasa de aprendizaje inicial y k es el número de la iteración actual. τ es el **parámetro de decaimiento**, que controla cuán

rápido disminuye α.

Una heurística útil para definir el parámetro de decaimiento es vincularlo al número total de iteraciones planificadas:

$$\tau = \frac{\alpha_0}{N_{\text{máx}}}$$

Supongamos que queremos encontrar el mínimo de la siguiente función: $f(x, y) = (x-2)^2+(y+3)^2$.

[9]:
```python
# Definimos la función objetivo y su gradiente
def f_multi(x):
    return (x[0] - 2) ** 2 + (x[1] + 3) ** 2

def grad_f_multi(x):
    dx = 2 * (x[0] - 2)
    dy = 2 * (x[1] + 3)
    return np.array([dx, dy])

# Algoritmo de descenso de gradiente
def dg_tau(x_actual, alpha_0, tau, max_iter, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        alpha_k = alpha_0 / (1 + i * tau)
        x_nuevo = x_actual - alpha_k * grad_f_multi(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_f_multi(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial

# Parámetros del algoritmo
max_iter = 1000
x_actual = np.array([-5, 5])

# Tasa de aprendizaje inicial
alpha_0 = 1

# Constante de decaimiento
tau = 0.01

# Ejecución del algoritmo y resultados
r_tau = dg_tau(x_actual, alpha_0, tau, max_iter)

print("(x, y) =", r_tau[0])
print("f(x, y) =", f_multi(r_tau[0]))
```

```
print("Iteraciones =", len(r_tau[1]))
```

```
(x, y) = [ 2.00000033 -3.00000037]
f(x, y) = 2.459381416434157e-13
Iteraciones = 42
```

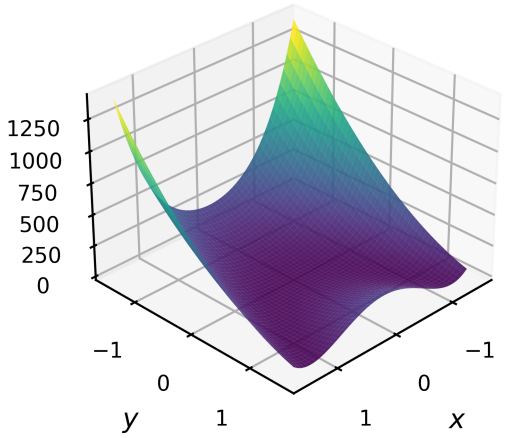
[10]:
```
# ------------------------------------------------
# Encontrar el mínimo de la función de Rosenbrock:
# f(x, y) = (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
# ------------------------------------------------

# Definimos la función objetivo y su gradiente
def rosenbrock(x):
    return (1 - x[0]) ** 2 + 100 * (x[1] - x[0] ** 2) ** 2

def grad_rosenbrock(x):
    dx = -2 * (1 - x[0]) - 400 * x[0] * (x[1] - x[0] ** 2)
    dy = 200 * (x[1] - x[0] ** 2)
    return np.array([dx, dy])

# Algoritmo de descenso de gradiente
def dg_rosenbrock(x_actual, alpha_0, tau, max_iter, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        alpha_k = alpha_0 / (1 + i * tau)
        x_nuevo = x_actual - alpha_k * grad_rosenbrock(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_rosenbrock(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial

# Parámetros del algoritmo
max_iter = 50000
x_actual = np.array([-1, -1])

# Tasa de aprendizaje inicial
alpha_0 = 0.002

# Constante de decaimiento
tau = alpha_0 / max_iter

# Ejecución del algoritmo y resultados
r_rosenbrock = dg_rosenbrock(x_actual, alpha_0, tau, max_iter)
```

```
print("(x, y) =", r_rosenbrock[0])
print("f(x, y) =", rosenbrock(r_rosenbrock[0]))
print("Iteraciones =", len(r_rosenbrock[1]))
```

```
(x, y) = [0.99944186 0.9988818 ]
f(x, y) = 3.1201517576786573e-07
Iteraciones = 7780
```

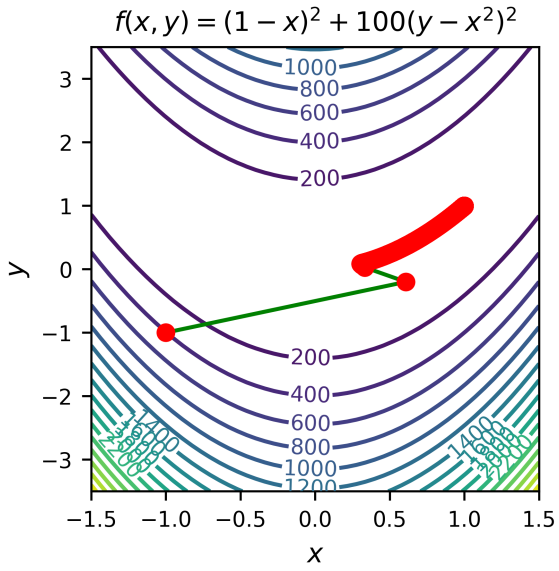
[11]:
```
# Graficar la función
x = np.linspace(-1.5, 1.5, 50)
y = np.linspace(-1.5, 1.5, 50)
X, Y = np.meshgrid(x, y)
Z = rosenbrock([X, Y])

fig = plt.figure(figsize = (3, 3), dpi = resolucion)
ax = fig.add_subplot(111, projection = "3d")
ax.plot_surface(X, Y, Z, cmap = "viridis", alpha = 0.9)
ax.view_init(elev = 30, azim = 45)

plt.title("$f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$", fontsize = 10)
ax.set_xlabel("$x$", fontsize = 10)
ax.set_ylabel("$y$", fontsize = 10)
ax.tick_params(labelsize = 8)
plt.show()
```

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$



[12]:
```
# Graficar la función
x = np.linspace(-1.5, 1.5, 50)
y = np.linspace(-3.5, 3.5, 50)
X, Y = np.meshgrid(x, y)
```

```
Z = rosenbrock([X, Y])

plt.figure(figsize = (3, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")

plt.clabel(contour, inline = 1, fontsize = 8)
plt.title("$f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$y$", fontsize = 10)

# Puntos de trayectoria
plt.plot(r_rosenbrock[1][:, 0], r_rosenbrock[1][:, 1], color = "green", zorder␣
 ↪= 2)
plt.scatter(r_rosenbrock[1][:, 0], r_rosenbrock[1][:, 1], color = "red", zorder␣
 ↪= 3)
plt.tick_params(labelsize = 8)
plt.show()
```



7 Método de descenso de gradiente con momento

El **método de descenso de gradiente con momento** es una técnica de optimización que acelera el descenso de gradientes convencional. Acumula un promedio ponderado de los gradientes pasados y lo usa para determinar la dirección del siguiente paso. Esto ayuda a suavisar las oscilaciones y a navegar más rápidamente por regiones con poca curvatura, como los valles de la función de Rosenbrock.

La idea es introducir una variable de "velocidad" (v) que acumula la dirección del gradiente en cada

paso. Esta velocidad se actualiza en cada iteración y luego se utiliza para actualizar los parámetros.

Para aplicar este método, se comienza calculando el gradiente de la función f en el punto actual $(\mathbf{x}_k)$:

$$\nabla f(\mathbf{x}_k)$$

Luego, se **actualiza la velocidad** combinando la velocidad anterior con el gradiente actual:

$$\mathbf{v}_{k+1} = \gamma \mathbf{v}_k + \alpha \nabla f(\mathbf{x}_k)$$

donde γ es el **coeficiente de momento** (un valor cercano a 1, comúnmente 0.90). Controla cuánta de la velocidad anterior se conserva. α es la **tasa de aprendizaje** (un valor pequeño, como 0.001).

Finalamente, se **actualizan las variables** en la dirección de la nueva velocidad:

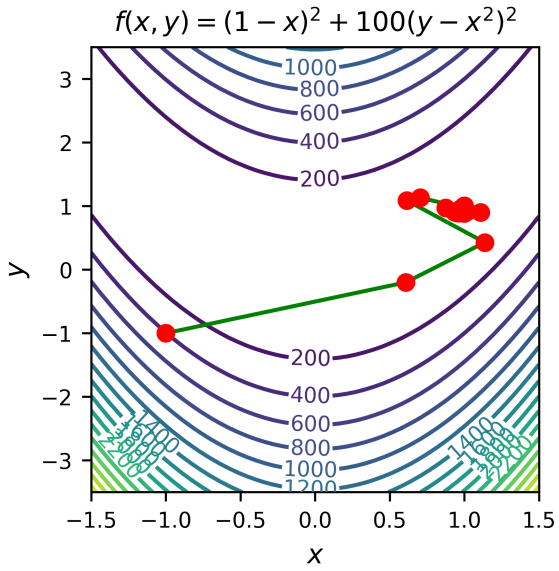
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{v}_{k+1}$$

[13]:
```python
# ------------------------------------------------
# Encontrar el mínimo de la función de Rosenbrock:
# f(x, y) = (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
# ------------------------------------------------

# Definimos la función objetivo y su gradiente
def rosenbrock(x):
    return (1 - x[0]) ** 2 + 100 * (x[1] - x[0] ** 2) ** 2

def grad_rosenbrock(x):
    dx = -2 * (1 - x[0]) - 400 * x[0] * (x[1] - x[0] ** 2)
    dy = 200 * (x[1] - x[0] ** 2)
    return np.array([dx, dy])

# Algoritmo de descenso de gradiente con momento
def dgm_rosenbrock(x_actual, gamma, alpha, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    velocidad = np.zeros(2)
    for i in range(max_iter):
        velocidad = gamma * velocidad + alpha * grad_rosenbrock(x_actual)
        x_nuevo = x_actual - velocidad
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_rosenbrock(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial
```

```
# Parámetros del algoritmo
x_actual = np.array([-1, -1])

# Tasa de aprendizaje
alpha = 0.002

# Coeficiente de momento
gamma = 0.5

# Ejecución del algoritmo y resultados
r_dgm_rosenbrock = dgm_rosenbrock(x_actual, gamma, alpha)

print("(x, y) =", r_dgm_rosenbrock[0])
print("f(x, y) =", rosenbrock(r_dgm_rosenbrock[0]))
print("Iteraciones =", len(r_dgm_rosenbrock[1]))
```

```
(x, y) = [0.99972157 0.99944211]
f(x, y) = 7.764506150442247e-08
Iteraciones = 3234
```

[14]:
```
# Graficar la función
x = np.linspace(-1.5, 1.5, 50)
y = np.linspace(-3.5, 3.5, 50)
X, Y = np.meshgrid(x, y)
Z = rosenbrock([X, Y])

plt.figure(figsize = (3, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")

plt.clabel(contour, inline = 1, fontsize = 8)
plt.title("$f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$y$", fontsize = 10)

# Puntos de trayectoria
plt.plot(r_dgm_rosenbrock[1][:, 0], r_dgm_rosenbrock[1][:, 1], color = "green",␣
 ↪zorder = 2)
plt.scatter(r_dgm_rosenbrock[1][:, 0], r_dgm_rosenbrock[1][:, 1], color =␣
 ↪"red", zorder = 3)
plt.tick_params(labelsize = 8)
plt.show()
```



8 Aplicaciones

8.1 Regresión lineal simple

En ingeniería de hardware, predecir el **consumo de energía** es crucial para el diseño de sistemas de refrigeración y la gestión de la eficiencia energética. Se sabe que la **frecuencia de reloj** (velocidad) de una CPU impacta directamente su consumo de energía; a mayor velocidad, mayor es la demanda energética.

Objetivo:

Construir un modelo de regresión lineal simple que permita predecir el consumo de energía de una CPU (Watt) a partir de su frecuencia de reloj (GHz).

Definición de Variables:

Para este modelo, se definen las siguientes variables:

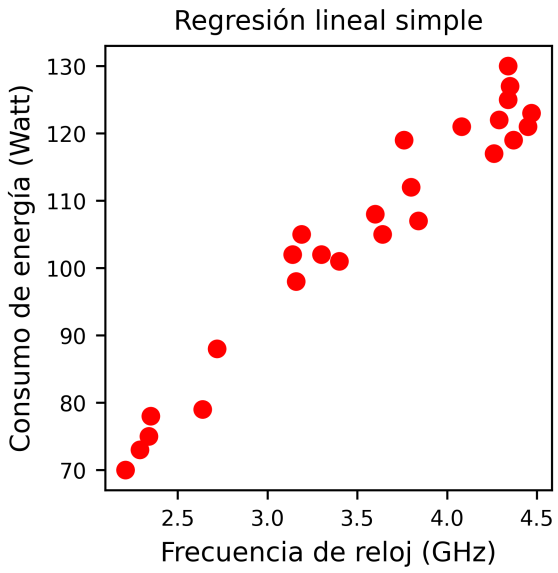
- **Variable independiente** (X): Frecuencia de Reloj (GHz). Es la variable que usaremos para predecir.
- **Variable dependiente** (Y): Consumo de Energía (Watt). Es la variable que queremos predecir.

El objetivo final es encontrar la ecuación de la recta:

$$Y = \beta_0 + \beta_1 x$$

que mejor describa esta relación y nos permita hacer estimaciones precisas.

[15]:
```
# Datos
frecuencia_reloj = np.array([4.29, 4.34, 2.72, 4.08, 3.60,
                             3.30, 3.84, 2.34, 3.64, 3.76,
                             3.14, 3.80, 4.34, 2.64, 3.16,
                             4.35, 4.45, 2.29, 3.19, 3.40,
                             4.26, 2.35, 4.47, 4.37, 2.21])

consumo_energia = np.array([122, 130,  88, 121, 108,
                            102, 107,  75, 105, 119,
                            102, 112, 125,  79,  98,
                            127, 121,  73, 105, 101,
                            117,  78, 123, 119, 70])

# Diagrama de dispersión
plt.figure(figsize = (3, 3), dpi = resolucion)
plt.scatter(frecuencia_reloj, consumo_energia, color = "red")
plt.xlabel("Frecuencia de reloj (GHz)", fontsize = 10)
plt.ylabel("Consumo de energía (Watt)", fontsize = 10)
plt.title("Regresión lineal simple", fontsize = 10)
plt.tick_params(labelsize = 8)
plt.show()
```



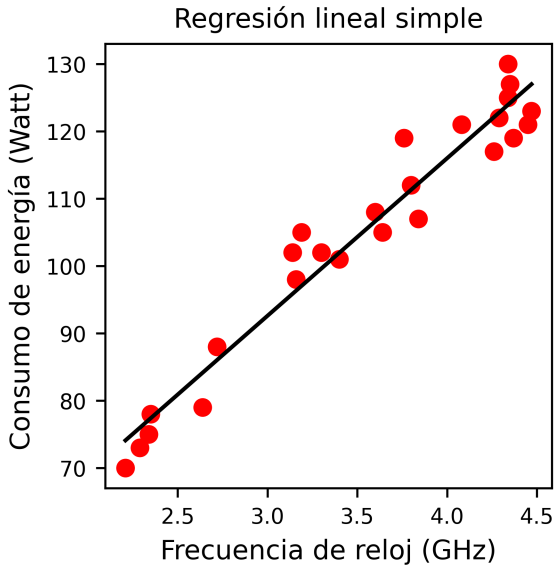
[16]:
```
# ------------------------------------------
# Encontrar el mínimo de la función:
# f(beta) = sum((y - beta_0 - beta_1 * x) ** 2)
# ------------------------------------------
```

```
# Definimos la función objetivo y su gradiente
def f_objetivo(beta, x, y):
    e = y - beta[0] - beta[1] * x
    return np.sum(e ** 2)

def grad_f_objetivo(beta, x, y):
    e = y - beta[0] - beta[1] * x
    d_beta_0 = -2 * np.sum(e)
    d_beta_1 = -2 * np.sum(e * x)
    return np.array([d_beta_0, d_beta_1])

# Algoritmo de descenso de gradiente
def dg_rls(beta_actual, x, y, alpha, max_iter = 10000, tol = 1e-6):
    beta_historial = np.array([beta_actual])
    for i in range(max_iter):
        beta_nuevo = beta_actual - alpha * grad_f_objetivo(beta_actual, x, y)
        beta_historial = np.vstack((beta_historial, beta_nuevo))
        criterio_1 = np.linalg.norm(grad_f_objetivo(beta_nuevo, x, y))
        criterio_2 = np.linalg.norm(beta_nuevo - beta_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        beta_actual = beta_nuevo
    return beta_nuevo, beta_historial

# Ajuste de parámetros del algoritmo
alpha = 0.001
x = frecuencia_reloj
y = consumo_energia
beta_actual = np.array([0, 1])

# Ejecución del algoritmo y resultados
r_rls = dg_rls(beta_actual, x, y, alpha)
beta_estimado = r_rls[0]

print("Beta estimado =", beta_estimado)
print("Iteraciones =", len(r_rls[1]))
```

```
Beta estimado = [22.38189069 23.40600308]
Iteraciones = 5150
```

[17]:
```
# Diagrama de dispersión
plt.figure(figsize = (3, 3), dpi = resolucion)
plt.scatter(frecuencia_reloj, consumo_energia, color = "red")
plt.xlabel("Frecuencia de reloj (GHz)", fontsize = 10)
plt.ylabel("Consumo de energía (Watt)", fontsize = 10)
plt.title("Regresión lineal simple", fontsize = 10)
```

```python
# Recta de regresión
x_regresion = np.linspace(min(frecuencia_reloj), max(frecuencia_reloj), 100)
y_regresion = beta_estimado[0] + beta_estimado[1] * x_regresion
plt.plot(x_regresion, y_regresion, color = "black")
plt.tick_params(labelsize = 8)
plt.show()
```



9 Ejercicios

Minimice la función $f(x) = (x-2)^3$.

Minimice la función $f(x, y) = \sin(x) + \cos(y)$.

Minimice la función de Beale $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$.

Minimice la función Himmelblau $f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$.

Minimice la función $f(x, y, z) = x^2 + y^2 + z^2$.

Minimice la función $f(x, y, z) = (x-1)^2 + (y-2)^2 + (z-3)^2 + \sin(x)\cos(y)\exp(z)$.

Juan F. Olivares Pacheco (jfolivar@uda.cl)

Universidad de Atacama, Facultad de Ingeniería, Departamento de Matemática