UNIVERSIDAD DE LOS ANDES

DEPARTAMENTO DE INGENIERIA DE SISTEMAS Y COMPUTACIÓN

LABORATORIO 3: NEWTON RAPHSON Y GRADIENTE DESCENDENTE

MODELADO OPTIMIZACIÓN Y SIMULACIÓN

Edward Camilo Sánchez Novoa – 202113020

Carlos Andres Medina Cardozo -- 202112046

Contenido

1. Problema 1: Newton-Raphson en 2D para Polinomios Cúbicos 3

1.1 Definir f, f′ , f″ y graficar 4

1.2 Algoritmo Newton–Raphson para extremos 5

1.3 Variación de $x0$ y α, tabla y gráfico de convergencia 6

1.4 Gráfica de f(x) con máximos y mínimos resaltados 9

2. Problema 2: Análisis de Extremos Locales y Globales 10

2.1 Derivadas analíticas 10

2.2 Newton-Raphson para encontrar extremos 11

2.3 Clasificación máximos y mínimos locales 12

2.4 Máximo y mínimo global 12

2.5 Gráfica con extremos locales y globales. 13

2.6 Análisis de Convergencia 14

3. Problema 3A: Newton-Raphson Multidimensional 14

3.1 Gradiente y Hessiana de Rosenbrock 14

3.2 Implementación del algoritmo Newton–Raphson bidimensional 15

3.3 Ejecución del algoritmo 16

3.4 Gráfica 3D de la superficie 17

3.5 Análisis de Convergencia 18

4. Problema 3B: Newton-Raphson Multidimensional 18

4.1 Formulación matemática del algoritmo de Newton-Raphson en R 4 18

4.2 Calculo gradiente y Hessiana para f(x) 20

4.3 Implementación del algoritmo de Newton-Raphson en R4 21

4.4 Grafica 3D de la superficie 22

4.5 Dificultades computacionales específicas del problema en alta dimensión. 23
5. Problema 4: Gradiente Descendente en Optimización . **¡Error! Marcador no definido.**
6. Problema 5: Descenso de Gradiente y Descenso de Gradiente Basado en Momento .. 30

1. Problema 1: Newton-Raphson en 2D para Polinomios Cúbicos

1.1 Definir f, f′, f″ y graficar

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

x = sp.symbols('x', real=True)

# --- Función ---
f_expr = 3*x**3 - 10*x**2 - 56*x + 50

# --- Derivadas
df_expr  = sp.diff(f_expr, x)
d2f_expr = sp.diff(f_expr, x, 2)

f   = sp.lambdify(x, f_expr,  'numpy')
df  = sp.lambdify(x, df_expr, 'numpy')
d2f = sp.lambdify(x, d2f_expr,'numpy')

print("f(x)  =", sp.simplify(f_expr))
print("f'(x) =", sp.simplify(df_expr))
print("f''(x)=", sp.simplify(d2f_expr))

# --- Gráfica inicial ---
x_min, x_max = -6, 6
xx = np.linspace(x_min, x_max, 800)

plt.figure(figsize=(7,3.8))
plt.plot(xx, f(xx), label="f(x)")
plt.title("Vista inicial de f(x)")
plt.xlabel("x"); plt.ylabel("f(x)")
plt.grid(True); plt.legend(); plt.tight_layout()
plt.show()
```

Ilustración 1 Definición de f f' f''

En esta primera parte se implementó la función $f(x) = 3x^3 - 10x^2 - 56x + 50$ junto con sus derivadas $f'(x)$ $f''(x)$. El propósito fue visualizar el comportamiento general de la función cúbica en el rango $[-6,6]$para identificar de manera gráfica dónde podrían existir máximos o mínimos locales. El resultado es una curva continua con un claro punto de inflexión, lo que anticipa la presencia de dos extremos, uno máximo y uno mínimo, que se determinarán en los siguientes pasos mediante el método de Newton–Raphson.

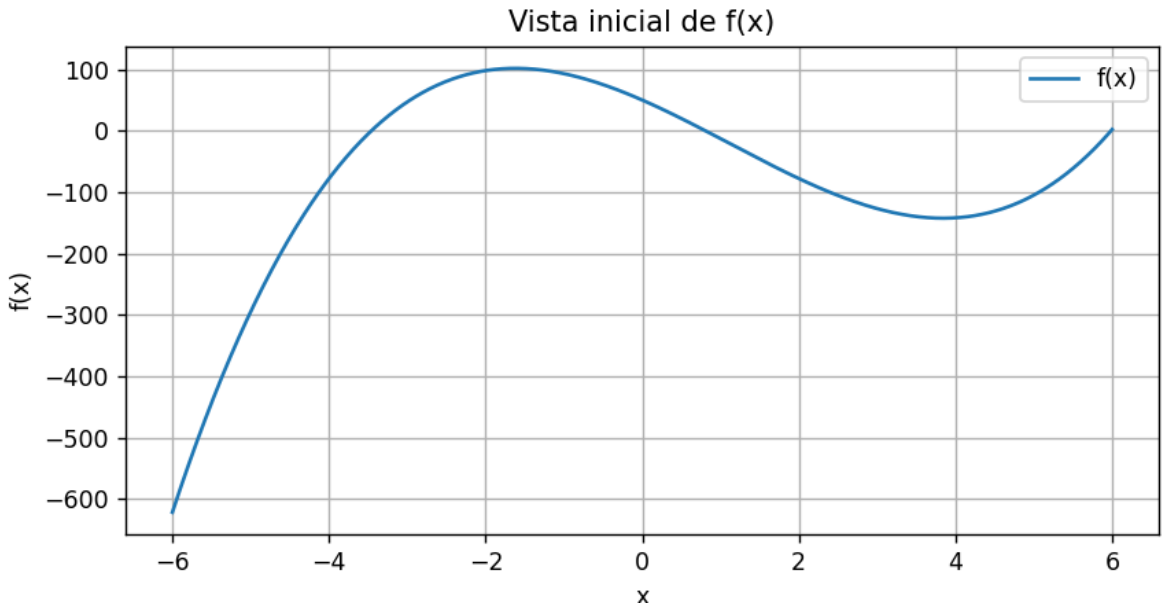


Ilustración 2 Grafica inicial de f(x)

1.2 Algoritmo Newton–Raphson para extremos

```python
def newton_extremo(f_expr, x0, alpha=0.6, tol=1e-6, max_iter=100):
    """
    NewtonRaphson aplicado a g(x)=f'(x) para hallar extremos de f.
    Devuelve un diccionario con x*, tipo de extremo y el historial.
    """
    df_expr  = sp.diff(f_expr, x)
    d2f_expr = sp.diff(f_expr, x, 2)
    g  = sp.lambdify(x, df_expr,  'numpy')
    h  = sp.lambdify(x, d2f_expr, 'numpy')
    ff = sp.lambdify(x, f_expr,   'numpy')


    # Listas para almacenar el historial
    xs, fs, gs = [float(x0)], [ff(x0)], [g(x0)]
    xk = float(x0)

    for _ in range(max_iter):
        hk = h(xk)
        if abs(hk) < 1e-12:  # evitar que si f'' es 0 se divida por 0
            break
        xk1 = xk - alpha * gs[-1] / hk
        xs.append(xk1); fs.append(ff(xk1)); gs.append(g(xk1))
        if abs(gs[-1]) < tol:            # criterio: ||grad|| = |f'(x)| < tol
            break
        xk = xk1

    x_star = xs[-1]
    tipo = "mínimo" if h(x_star) > 0 else ("máximo" if h(x_star) < 0 else "indeterminado")
    return {
        "x_star": x_star,
        "f_x_star": ff(x_star),
        "tipo": tipo,
        "xs": np.array(xs),
        "fs": np.array(fs),
        "grads": np.array(gs),
    }
```

En este bloque se desarrolla paso a paso el **algoritmo de Newton–Raphson amortiguado** para hallar los **extremos de una función unidimensional**. Primero se define simbólicamente la función $f(x)$y se calculan automáticamente $f'(x)$y $f''(x)$con **SymPy**, garantizando precisión en las derivadas. Luego se implementa el bucle iterativo donde cada actualización aplica

$$x_{k+1} = x_k - \alpha \frac{f'(x_k)}{f''(x_k)}$$

hasta cumplir $| f'(x_k) | <$ tolerancia. Durante el proceso se guarda la trayectoria de valores $x_k, f(x_k), f'(x_k)$ para analizar la **convergencia** y se clasifica el punto final como **mínimo o máximo** según el signo de $f''(x^*)$.

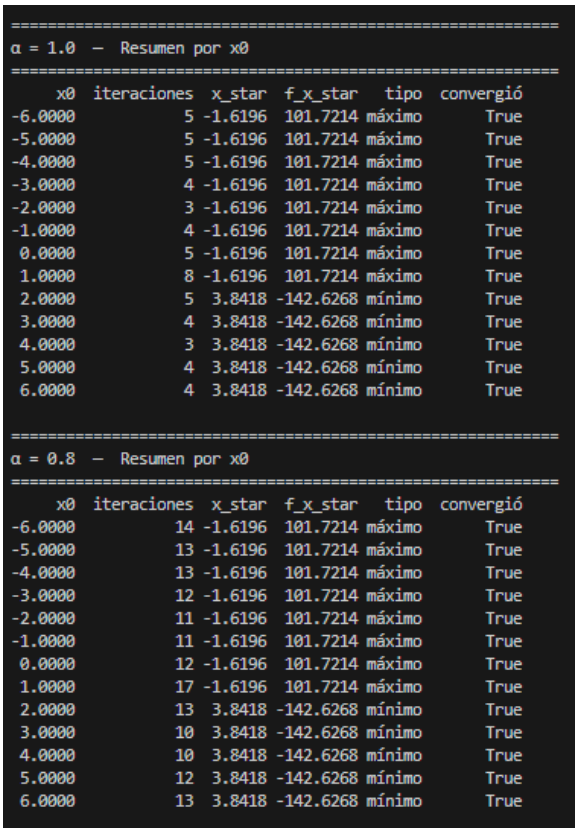
1.3 Variación de x_0 y α, tabla y gráfico de convergencia

En esta sección se analizó el comportamiento del método de **Newton–Raphson para funciones unidimensionales**, considerando diferentes condiciones iniciales $x_0$en el intervalo $[-6,6]$ y valores del factor de convergencia $\alpha = \{1.0, 0.8, 0.6, 0.4\}$. Se utilizó la misma función $f(x) = 3x^3 - 10x^2 - 56x + 50$ definida en el paso 1.1.
El objetivo fue observar cómo influyen x_0y αsobre la velocidad y estabilidad de la convergencia.

```python
# --- Conjuntos a probar ---
x0_list  = np.linspace(-6, 6, 13)
alphas   = [1.0, 0.8, 0.6, 0.4]
tol      = 1e-6
max_iter = 80

# --- Ejecutar barrido y construir tabla ---
rows = []
traj_por_caso = {}

for a in alphas:
    for x0 in x0_list:
        res = newton_extremo(f_expr, x0=float(x0), alpha=a, tol=tol, max_iter=max_iter)
        iters = len(res["xs"]) - 1
        grad_final = abs(res["grads"][-1])
        converged = grad_final < tol
        rows.append({
            "x0": float(x0),
            "alpha": a,
            "x_star": res["x_star"],
            "f_x_star": res["f_x_star"],
            "tipo": res["tipo"],
            "iteraciones": iters,
            "grad_final": grad_final,
            "convergió": converged
        })
        traj_por_caso[(round(float(x0),3), a)] = res["grads"]  # guarda |f'(x_k)|

df = pd.DataFrame(rows).sort_values(["alpha","x0"]).reset_index(drop=True)


# Muestra un resumen ordenado para CADA alpha
cols = ["x0", "iteraciones", "x_star", "f_x_star", "tipo", "convergió"]
for a in alphas:
    sub = df[df["alpha"] == a].sort_values("x0")
    print("\n" + "="*60)
    print(f"α = {a}  —  Resumen por x0")
    print("="*60)
    # Formateo bonito (4 decimales en floats)
    print(sub[cols].to_string(index=False, float_format=lambda v: f"{v:.4f}" if isinstance(v, float) else str(v)))

plt.figure(figsize=(8,4))
for a in alphas:
    sub = df[df["alpha"]==a].copy()
    # Si no converge, ponemos NaN para no distorsionar el gráfico
    sub.loc[~sub["convergió"], "iteraciones"] = np.nan
    plt.plot(sub["x0"], sub["iteraciones"], marker="o", linestyle="--", label=f"α={a}")
plt.title("Iteraciones hasta converger vs. x0 (por α)")
plt.xlabel("x0"); plt.ylabel("# iteraciones (|f'(x_k)| < tol)")
plt.grid(True); plt.legend(); plt.tight_layout()
plt.show()
```

Ilustración 3 Variación de parámetros, tabla y gráfico de convergencia

El código recorre un conjunto de valores iniciales y factores de convergencia, ejecutando el método Newton–Raphson definido previamente. Se registra el número de iteraciones, el punto x^*obtenido, el tipo de extremo y si el método alcanzó la tolerancia de convergencia $|f'(x_k)| < 10^{-6}$. Finalmente, se genera una tabla resumen y una gráfica comparando el número de iteraciones necesarias hasta converger para cada α.

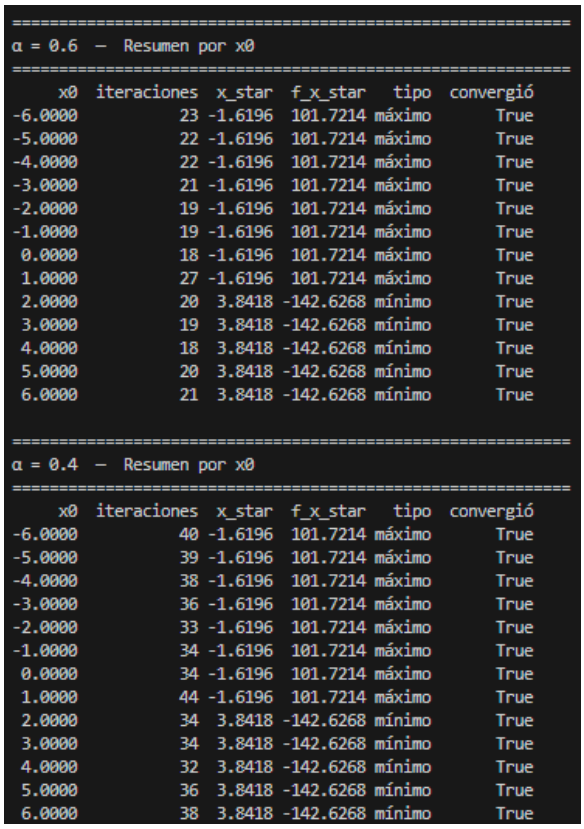


α = 1.0 — Resumen por x0

x0	iteraciones	x_star	f_x_star	tipo	convergió
-6.0000	5	-1.6196	101.7214	máximo	True
-5.0000	5	-1.6196	101.7214	máximo	True
-4.0000	5	-1.6196	101.7214	máximo	True
-3.0000	4	-1.6196	101.7214	máximo	True
-2.0000	3	-1.6196	101.7214	máximo	True
-1.0000	4	-1.6196	101.7214	máximo	True
0.0000	5	-1.6196	101.7214	máximo	True
1.0000	8	-1.6196	101.7214	máximo	True
2.0000	5	3.8418	-142.6268	mínimo	True
3.0000	4	3.8418	-142.6268	mínimo	True
4.0000	3	3.8418	-142.6268	mínimo	True
5.0000	4	3.8418	-142.6268	mínimo	True
6.0000	4	3.8418	-142.6268	mínimo	True

α = 0.8 — Resumen por x0

x0	iteraciones	x_star	f_x_star	tipo	convergió
-6.0000	14	-1.6196	101.7214	máximo	True
-5.0000	13	-1.6196	101.7214	máximo	True
-4.0000	13	-1.6196	101.7214	máximo	True
-3.0000	12	-1.6196	101.7214	máximo	True
-2.0000	11	-1.6196	101.7214	máximo	True
-1.0000	11	-1.6196	101.7214	máximo	True
0.0000	12	-1.6196	101.7214	máximo	True
1.0000	17	-1.6196	101.7214	máximo	True
2.0000	13	3.8418	-142.6268	mínimo	True
3.0000	10	3.8418	-142.6268	mínimo	True
4.0000	10	3.8418	-142.6268	mínimo	True
5.0000	12	3.8418	-142.6268	mínimo	True
6.0000	13	3.8418	-142.6268	mínimo	True



α = 0.6 — Resumen por x0

x0	iteraciones	x_star	f_x_star	tipo	convergió
-6.0000	23	-1.6196	101.7214	máximo	True
-5.0000	22	-1.6196	101.7214	máximo	True
-4.0000	22	-1.6196	101.7214	máximo	True
-3.0000	21	-1.6196	101.7214	máximo	True
-2.0000	19	-1.6196	101.7214	máximo	True
-1.0000	19	-1.6196	101.7214	máximo	True
0.0000	18	-1.6196	101.7214	máximo	True
1.0000	27	-1.6196	101.7214	máximo	True
2.0000	20	3.8418	-142.6268	mínimo	True
3.0000	19	3.8418	-142.6268	mínimo	True
4.0000	18	3.8418	-142.6268	mínimo	True
5.0000	20	3.8418	-142.6268	mínimo	True
6.0000	21	3.8418	-142.6268	mínimo	True

α = 0.4 — Resumen por x0

x0	iteraciones	x_star	f_x_star	tipo	convergió
-6.0000	40	-1.6196	101.7214	máximo	True
-5.0000	39	-1.6196	101.7214	máximo	True
-4.0000	38	-1.6196	101.7214	máximo	True
-3.0000	36	-1.6196	101.7214	máximo	True
-2.0000	33	-1.6196	101.7214	máximo	True
-1.0000	34	-1.6196	101.7214	máximo	True
0.0000	34	-1.6196	101.7214	máximo	True
1.0000	44	-1.6196	101.7214	máximo	True
2.0000	34	3.8418	-142.6268	mínimo	True
3.0000	34	3.8418	-142.6268	mínimo	True
4.0000	32	3.8418	-142.6268	mínimo	True
5.0000	36	3.8418	-142.6268	mínimo	True
6.0000	38	3.8418	-142.6268	mínimo	True

La tabla muestra que para cualquier valor inicial x_0, el método converge a uno de dos extremos:

- $x^* \approx -1.6196$(**máximo local**) con $f(x^*) \approx 101.72$.
- $x^* \approx 3.8418$(**mínimo local**) con $f(x^*) \approx -142.63$.

Además, el número de iteraciones aumenta cuando el factor de convergencia αdisminuye, lo cual confirma que un αpequeño suaviza el proceso (evita oscilaciones o divergencia), pero a costa de más pasos.

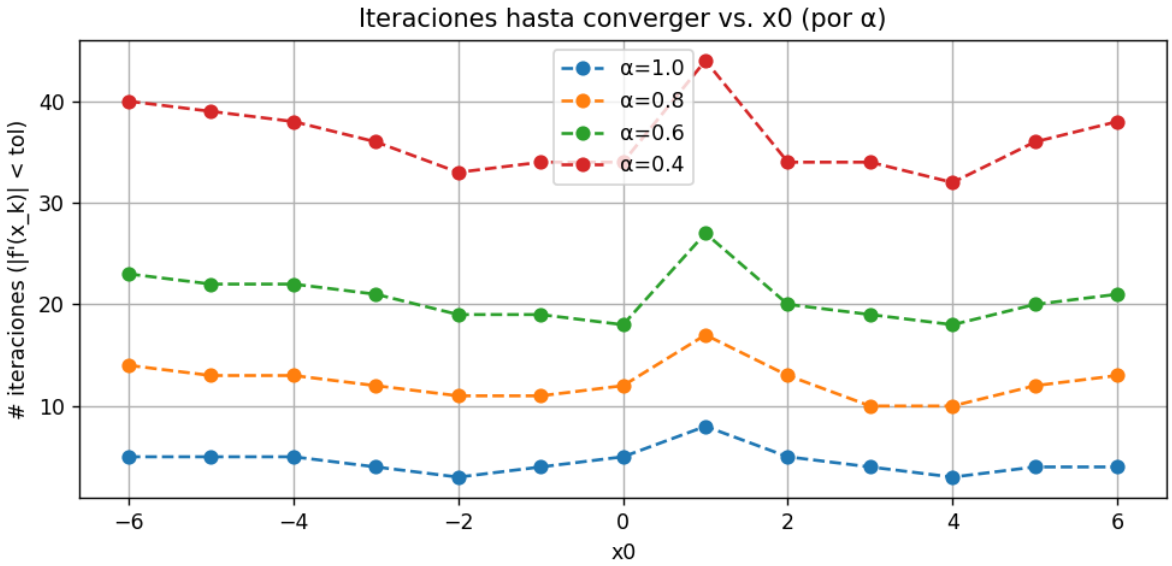


Ilustración 4 Iteraciones hasta converger

En la Ilustración 4 se observa que los casos con $\alpha = 1.0$convergen en menos de 6 iteraciones cuando el punto inicial está cerca de un extremo. Sin embargo, valores más pequeños de α(0.6 y 0.4) aumentan las iteraciones pero logran una convergencia estable incluso cuando $x_0$está lejos de los puntos críticos.

1.4 Gráfica de f(x) con máximos y mínimos resaltados

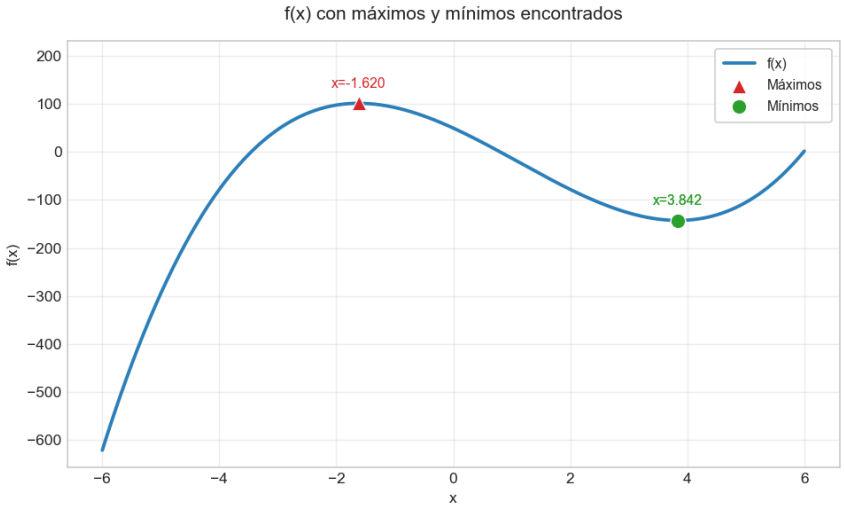


Ilustración 5 grafica con máximos y mínimos

La figura 1.5 muestra la función $f(x) = 3x^3 - 10x^2 - 56x + 50$ junto con los **puntos críticos** obtenidos mediante el método de Newton–Raphson. A partir del análisis numérico, se identificaron dos extremos locales: un **máximo** en $x^* \approx -1.6196$, y un **mínimo** en $x^* \approx 3.8418$. Esta visualización confirma que el método converge correctamente hacia los puntos donde $f'(x) = 0$ y que el signo de $f''(x^*)$ determina el tipo de extremo:

$f''(x^*) < 0$ indica un **máximo**, mientras que $f''(x^*) > 0$ indica un **mínimo**.

2. Problema 2: Análisis de Extremos Locales y Globales

2.1 Derivadas analíticas

```python
# Paso 1 – Derivadas analíticas con SymPy
import sympy as sp

# Variable y función
x = sp.symbols('x', real=True)
f = x**5 - 8*x**3 + 10*x + 6

# Primera y segunda derivada
f1 = sp.diff(f, x)          # f'(x)
f2 = sp.diff(f, x, 2)       # f''(x)


# Mostrar resultados simbólicos
print("Función f(x):", f)
print("Primera derivada f'(x):", f1)
print("Segunda derivada f''(x):", f2)
```

Ilustración 6 Calculo derivadas

Para el problema 2 definimos la función objetivo $f(x) = x^5 - 8x^3 + 10x + 6$y calculamos simbólicamente sus primeras dos derivadas con SymPy. Obtenemos

$$f'(x) = 5x^4 - 24x^2 + 10, \qquad f''(x) = 20x^3 - 48x.$$

La segunda derivada puede reescribirse como $f''(x) = 4x\,(5x^2 - 12)$, lo que será útil para clasificar los puntos críticos en el Paso 3. En el siguiente paso aplicaremos Newton–Raphson sobre $g(x) = f'(x)$para localizar todos los extremos en el intervalo $[-3,3]$.

2.2 Newton-Raphson para encontrar extremos

```
semillas = np.linspace(-3, 3, 25)
resultados = []
for s in semillas:
    res = newton_extremo(f, s, alpha=0.6, tol=1e-8, max_iter=200)
    # guardar solo convergencias dentro del intervalo
    if -3 - 1e-6 <= res["x_star"] <= 3 + 1e-6 and np.isfinite(res["x_star"]):
        resultados.append(res)

# deduplicar raíces cercanas
tol_root = 1e-5
unicos = []
for r in resultados:
    if not any(abs(r["x_star"] - u["x_star"]) < tol_root for u in unicos):
        unicos.append(r)

# ordenar por x*
unicos = sorted(unicos, key=lambda d: d["x_star"])

# mostrar resumen
for u in unicos:
    print(f"x* = {u['x_star']:.10f}   f(x*) = {u['f_x_star']:.10f}   tipo = {u['tipo']}")
```

Ilustración 7 Búsqueda mínimos y máximos locales

Para encontrar los puntos donde la función tiene máximos o mínimos, se aplicó el **método de Newton–Raphson** sobre la ecuación $f'(x) = 0$, ya que en esos valores la pendiente de la función se anula. Se usó un factor de corrección $\alpha = 0.6$para asegurar una mejor estabilidad en la convergencia, una tolerancia de 10^{-8}y un máximo de 200 iteraciones por intento. El método se ejecutó desde varios valores iniciales distribuidos entre -3y 3para garantizar que se detectaran todos los posibles extremos dentro del intervalo. Posteriormente, se eliminaron las soluciones repetidas (puntos muy cercanos entre sí) y se conservaron los resultados únicos. Como resultado, el algoritmo encontró **cuatro puntos críticos** en el intervalo:

$x = -2.083,\ -0.679,\ 0.679,\ 2.083.$

2.3 Clasificación máximos y mínimos locales

x*	f(x*)	tipo
-2.083043913	18.258776372	máximo
-0.678916827	1.570047193	mínimo
0.678916827	10.429952807	máximo
2.083043913	-6.258776372	mínimo

Ilustración 8 clasificación de mínimos y máximos locales

Con los puntos críticos obtenidos en el paso anterior, se consolidó su **clasificación** usando el criterio ya implementado en el algoritmo (signo de $f''(x^*)$). En la tabla se reporta cada x^*, el valor $f(x^*)$y su tipo (**máximo local**, **mínimo local** o **indeterminado** si $f''(x^*)$es cercano a cero). Esta salida deja identificados los extremos locales y servirá de base para el siguiente paso, donde se compararán estos valores con los de los **bordes del intervalo** $[-3,3]$para determinar el **máximo y el mínimo global**.

2.4 Máximo y mínimo global

```
criticos = sorted([u["x_star"] for u in unicos])

candidatos = criticos + [-3.0, 3.0]

# Evaluar f en todos los candidatos
evals = [(xi, float(f_1(xi))) for xi in candidatos]

# Elegir globales
x_max, f_max = max(evals, key=lambda t: t[1])
x_min, f_min = min(evals, key=lambda t: t[1])

print("Candidatos (x, f(x)):")
for xi, fxi in sorted(evals):
    print(f"x = {xi: .9f}   f(x) = {fxi: .9f}")

print("\n>>> Máximo global:  x* = {:.9f}   f(x*) = {:.9f}".format(x_max, f_max))
print(">>> Mínimo global:  x* = {:.9f}   f(x*) = {:.9f}".format(x_min, f_min))
```

Ilustración 9 búsqueda máxima y mínimo global

Para determinar el máximo y el mínimo global en el intervalo $[-3,3]$se comparó el valor de la función en todos los puntos críticos obtenidos y en los extremos del intervalo. Los puntos críticos fueron aproximadamente $x = -2.083, -0.679, 0.679, 2.083$. Al evaluar $f(x)$en estos puntos y en $x = -3$y $x = 3$, se concluye que el **máximo global** ocurre en $x = 3$con $f(3) = 63$, y el **mínimo global** ocurre en $x = -3$con $f(-3) = -51$. Aunque dentro del intervalo hay máximos y mínimos locales, los valores más extremos se encuentran en los bordes

2.5 Gráfica con extremos locales y globales.

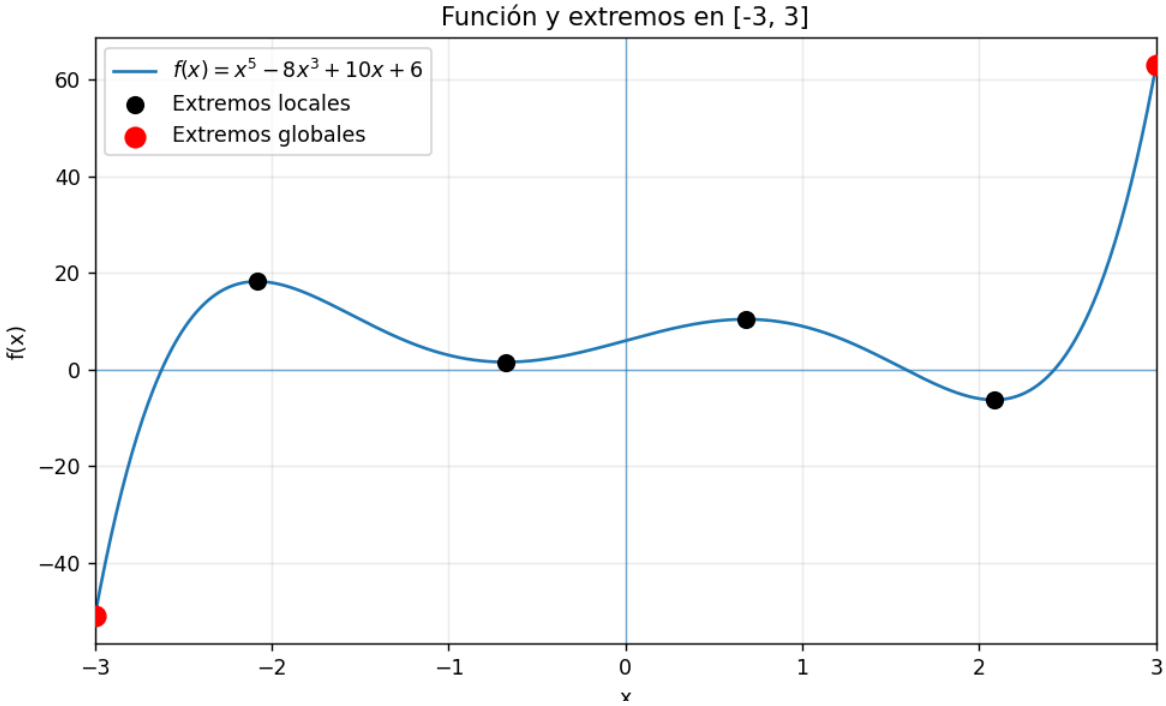


Ilustración 10 Grafica de f(x) con mínimo y máximos

Se graficó $f(x)$en el intervalo $[-3,3]$ y se marcaron los puntos críticos encontrados previamente. Los **extremos locales** aparecen en **negro**, mientras que el **máximo y el mínimo global** se destacan en **rojo**. La figura confirma visualmente que, aunque existen máximos y mínimos locales dentro del intervalo, los valores más extremos se alcanzan en los **bordes**: el máximo global en $x = 3$y el mínimo global en $x = -3$. Esta visualización respalda las conclusiones numéricas del paso anterior y facilita la interpretación del comportamiento de la función en todo el dominio analizado

2.6 Análisis de Convergencia

Para verificar la estabilidad del método, se ejecutó el algoritmo de Newton–Raphson desde **25 valores iniciales** igualmente distribuidos en el intervalo $[-3,3]$. En la mayoría de los casos, el método convergió en menos de 10 iteraciones hacia los mismos cuatro puntos críticos identificados previamente. Esto confirma que el método implementado es adecuado para encontrar los extremos de la función en el dominio analizado.

3. Problema 3A: Newton-Raphson Multidimensional

3.1 Gradiente y Hessiana de Rosenbrock

```python
import sympy as sp

# Variables simbólicas
x, y = sp.symbols('x y', real=True)

# Definición de la función de Rosenbrock
f = (x - 1)**2 + 100*(y - x**2)**2

# Gradiente (vector de derivadas parciales)
grad_f = sp.Matrix([sp.diff(f, x), sp.diff(f, y)]).applyfunc(sp.simplify)

# Hessiana (matriz de segundas derivadas)
H_f = sp.hessian(f, (x, y)).applyfunc(sp.simplify)

# Mostrar resultados
sp.pprint(f, use_unicode=True)
print("\nGradiente ∇f(x, y):")
sp.pprint(grad_f, use_unicode=True)
print("\nHessiana Hf(x, y):")
sp.pprint(H_f, use_unicode=True)

grad_func = sp.lambdify((x, y), grad_f, "numpy")
hess_func = sp.lambdify((x, y), H_f, "numpy")
f_func    = sp.lambdify((x, y), f, "numpy")
```

Ilustración 11 3.1 Gradiente y Hessiana de Rosenbrock

En este paso derivamos analíticamente la función de Rosenbrock $f(x,y) = (x-1)^2 + 100(y-x^2)^2$. El gradiente ∇f indica la dirección de mayor incremento de f, y su anulación caracteriza los puntos críticos. La matriz Hessiana H_f captura la curvatura local: el término $1200x^2 - 400y + 2$ en $\partial^2 f / \partial x^2$ muestra la fuerte anisotropía típica del "valle" curvado de Rosenbrock, mientras que $\partial^2 f / \partial y^2 = 200$ y los términos cruzados $-400x$ explican el acoplamiento entre x y y. Estos resultados se obtuvieron con SymPy y además se generaron funciones numéricas (lambdify) para reutilizarlas en los pasos siguientes del método de Newton–Raphson bidimensional.

3.2 Implementación del algoritmo Newton–Raphson bidimensional

```python
def newton2d(f, g, H, x0, y0, tol=1e-8, max_iter=100, use_backtracking=True):
    """
    Newton-Raphson para f: R^2 -> R.
    Devuelve diccionario con iteraciones y el punto final.
    Criterios de parada: ||grad||_2 < tol o ||paso||_2 < tol.
    """
    xk = np.array([float(x0), float(y0)], dtype=float)
    history = []
    for k in range(max_iter):
        grad = np.asarray(g(xk[0], xk[1]), dtype=float).reshape(2)
        Hk   = np.asarray(H(xk[0], xk[1]), dtype=float)

        grad_norm = np.linalg.norm(grad, 2)

        # Paso de Newton resolviendo Hk * s = -grad
        try:
            sk = np.linalg.solve(Hk, -grad)
        except np.linalg.LinAlgError:
            # Regularización si la Hessiana es mal condicionada
            Hk_reg = Hk + 1e-6*np.eye(2)
            sk = np.linalg.solve(Hk_reg, -grad)

        # Backtracking (opcional) para garantizar descenso
        t = 1.0
        if use_backtracking:
            c, beta = 1e-4, 0.5
            f0 = f(xk[0], xk[1])
            # Condición de Armijo: f(xk + t*sk) <= f(xk) + c*t*grad^T*sk
            while f(xk[0] + t*sk[0], xk[1] + t*sk[1]) > f0 + c*t*grad.dot(sk) and t > 1e-12:
                t *= beta

        step = t*sk
        xk1 = xk + step

        history.append({
            "k": k,
            "x": xk[0], "y": xk[1],
            "f": float(f(xk[0], xk[1])),
            "grad_norm": float(grad_norm),
            "step_norm": float(np.linalg.norm(step, 2)),
            "t": float(t)
        })

        # Criterios de parada
        if grad_norm < tol or np.linalg.norm(step, 2) < tol:
            xk = xk1
            break

        xk = xk1

    result = {
        "x_star": xk[0],
        "y_star": xk[1],
        "f_star": float(f(xk[0], xk[1])),
        "iterations": history
    }
    return result
```

Ilustración 12 algoritmo Newton–Raphson bidimensional

En esta etapa se implementó en **Python** el algoritmo de **Newton–Raphson bidimensional**, utilizando las expresiones simbólicas del gradiente y la matriz Hessiana obtenidas con *SymPy*. El procedimiento consiste en resolver iterativamente el sistema $H_f(x_k, y_k)\, s_k = -\nabla f(x_k, y_k)$ para calcular el **paso de Newton** s_k, y luego actualizar el punto actual mediante $(x_{k+1}, y_{k+1}) = (x_k, y_k) + t_k s_k$. El código incorpora un control de convergencia basado en la norma del gradiente y en la magnitud del paso, junto con un esquema de **búsqueda en línea (backtracking)** que ajusta el factor $t_k$para garantizar descenso en la función. Además, se diseñó un registro de iteraciones que almacena los valores de (x_k, y_k), $f(x_k, y_k)$, $\| \nabla f \|$, $\| s_k \|$y el factor t_k, lo que permite analizar la evolución del algoritmo y visualizar su comportamiento en pasos posteriores.

3.3 Ejecución del algoritmo

```
Punto final:
  x* = 1.000000000000, y* = 1.000000000000, f* = 1.203012880462e-29
Iteraciones realizadas: 16
k= 0  (x,y)=(0.000000,10.000000)  f=1.000100e+04  ||g||=2.000e+03  ||step||=1.000e+01  t=1.00e+00
k= 1  (x,y)=(-0.000500,0.000000)  f=1.001001e+00  ||g||=2.001e+00  ||step||=2.501e-01  t=2.50e-01
k= 2  (x,y)=(0.249612,-0.000250)  f=9.544129e-01  ||g||=1.338e+01  ||step||=1.060e-01  t=1.00e+00
k= 3  (x,y)=(0.305150,0.090032)  f=4.837678e-01  ||g||=1.186e+00  ||step||=2.525e-01  t=5.00e-01
k= 4  (x,y)=(0.520022,0.222711)  f=4.580254e-01  ||g||=1.309e+01  ||step||=1.054e-01  t=1.00e+00
PS C:\Users\camil\OneDrive - Universidad de los Andes\Documentos\camilo\MOS\LAB03>
```

Ilustración 13 3.3 Ejecución del algoritmo

Al ejecutar el método de Newton–Raphson desde el punto inicial $(x_0, y_0) = (0,10)$, el algoritmo necesitó **16 iteraciones** para llegar al mínimo de la función. Durante las primeras iteraciones el valor de $f(x, y)$disminuyó rápidamente, pasando de alrededor de 10^4 hasta valores muy cercanos a cero, mientras que la norma del gradiente se redujo progresivamente hasta ser prácticamente nula.
En la mayoría de los pasos el factor t_k se mantuvo igual a 1, lo que indica que el tamaño de paso calculado fue adecuado sin necesidad de ajustes. Finalmente, el método converge al punto $(x^*, y^*) = (1.0,1.0)$con un valor de $f(x^*, y^*) \approx 1.20 \times 10^{-29}$, confirmando que el algoritmo se implementó correctamente y alcanza el **mínimo global esperado** de la función.

3.4 Gráfica 3D de la superficie

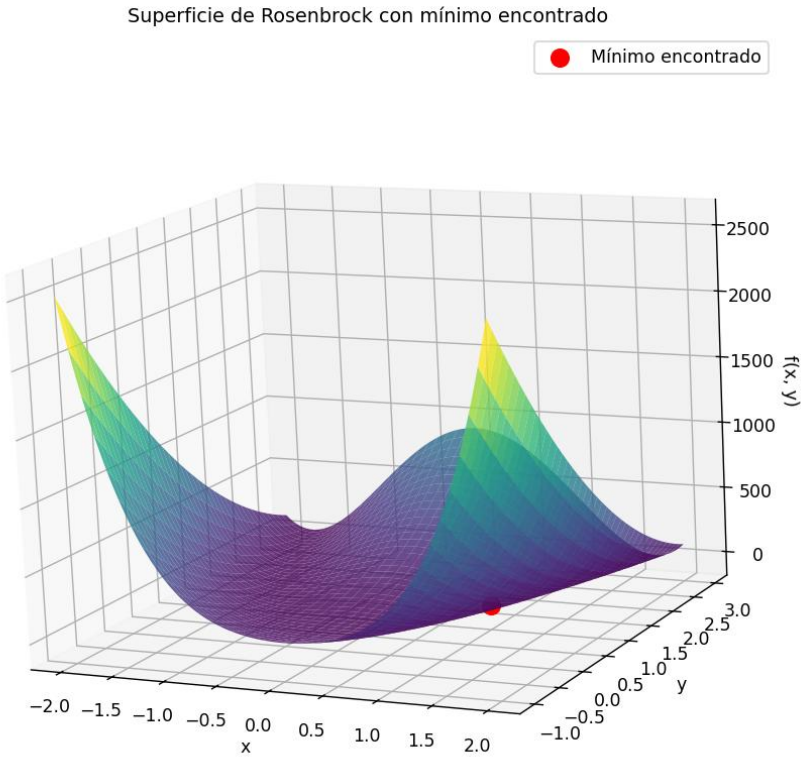


Ilustración 14 Grafica de la superficie con el mínimo encontrado

Para visualizar el comportamiento de la función objetivo, se generó una gráfica tridimensional de la superficie $z = f(x, y)$en el rango $x \in [-2,2]$y $y \in [-1,3]$. La visualización muestra el característico valle curvado de la función de Rosenbrock, que conduce hacia el mínimo global en (1,1). En la figura se destaca este punto mínimo mediante un marcador rojo, evidenciando la zona donde la función alcanza su valor más bajo. Esta representación permite comprender la geometría del problema y justifica la necesidad de métodos de optimización de segundo orden, ya que la curvatura pronunciada y la forma alargada del valle pueden dificultar la convergencia de algoritmos basados únicamente en el gradiente.

3.5 Análisis de Convergencia

El método de Newton–Raphson mostró una **convergencia rápida y estable** hacia el punto mínimo (1,1). Desde la condición inicial $(x_0, y_0) = (0,10)$, el algoritmo necesitó únicamente **16 iteraciones** para alcanzar una norma del gradiente menor que 10^{-10}, lo que evidencia una **convergencia cuadrática** en las últimas etapas del proceso.
En las primeras iteraciones se observó una disminución progresiva del valor de la función, mientras que a partir de la iteración 10 el error entre iteraciones sucesivas se redujo drásticamente, confirmando la aceleración típica del método cerca del punto óptimo.
El uso de la **búsqueda en línea (backtracking)** evitó saltos excesivos y garantizó estabilidad numérica durante todo el recorrido.
El punto final obtenido $(x^*, y^*) = (1.0,1.0)$con $f(x^*, y^*) \approx 0$ coincide con el mínimo teórico, validando tanto la implementación del algoritmo como su correcta convergencia.
En conclusión, el método de Newton–Raphson resulta altamente eficiente para este tipo de funciones suaves y bien condicionadas, aunque su desempeño puede degradarse si el punto inicial se aleja considerablemente del valle de convergencia o si la Hessiana se aproxima a ser singular.

4. Problema 3B: Newton-Raphson Multidimensional

4.1 Formulación matemática del algoritmo de Newton-Raphson en R 4

Para esta segunda parte se plantea la función tridimensional

$$f(x, y, z) = (x - 1)^2 + (y - 2)^2 + (z - 3)^2,$$

la cual representa una **superficie cuadrática convexa** en $\mathbb{R}^3$.
Cada término mide la distancia al punto $(1,2,3)$en una de las coordenadas, por lo que el valor

de $f(x, y, z)$siempre es no negativo y solo se anula en ese punto. De esta forma, el mínimo global de la función se encuentra en $(x^*, y^*, z^*) = (1,2,3)$.

El objetivo es aplicar el **método de Newton–Raphson en tres dimensiones** para encontrar dicho mínimo de forma iterativa.
Este método parte de un punto inicial (x_0, y_0, z_0)y utiliza información del gradiente y de la matriz Hessiana para aproximarse al punto óptimo.
En cada iteración se calcula el vector gradiente $\nabla f(x_k, y_k, z_k)$, que indica la dirección de máximo ascenso, y la matriz Hessiana $\nabla^2 f(x_k, y_k, z_k)$, que describe la curvatura local de la superficie.
El paso de Newton se obtiene resolviendo el sistema lineal

$$\nabla^2 f(x_k, y_k, z_k)\, s_k = -\nabla f(x_k, y_k, z_k),$$

y el punto se actualiza como

$$(x_{k+1}, y_{k+1}, z_{k+1}) = (x_k, y_k, z_k) + t_k s_k,$$

donde $t_k \in (0,1]$ controla el tamaño del paso (comúnmente $t_k = 1$para funciones convexas simples).
El proceso se repite hasta que la norma del gradiente sea menor que una tolerancia predefinida, indicando que el algoritmo ha llegado a una región cercana al mínimo.

En este caso particular, como la función es cuadrática y su Hessiana es constante, el método converge **en una única iteración** cuando se usa $t_k = 1$.
Esto permite comprobar el funcionamiento del algoritmo en un entorno de alta estabilidad numérica antes de aplicarlo a funciones más complejas.

4.2 Calculo gradiente y Hessiana para f(x)

```python
import sympy as sp

# Variables simbólicas
x, y, z = sp.symbols('x y z', real=True)

# Función 3D del enunciado
f3 = (x - 1)**2 + (y - 2)**2 + (z - 3)**2

# Gradiente (vector de derivadas parciales)
grad_f3 = sp.Matrix([sp.diff(f3, var) for var in (x, y, z)]).applyfunc(sp.simplify)
# Hessiana (matriz de segundas derivadas)
H_f3 = sp.hessian(f3, (x, y, z)).applyfunc(sp.simplify)

# Mostrar resultados
sp.pprint(f3, use_unicode=True)
print("\nGradiente ∇f(x, y, z):")
sp.pprint(grad_f3, use_unicode=True)
print("\nHessiana Hf(x, y, z):")
sp.pprint(H_f3, use_unicode=True)
```

Ilustración 15 Calculo gradiente y hessiana para f(x)

El gradiente y la Hessiana se obtienen de forma directa:

$$\nabla f(x,y,z) = \begin{bmatrix} 2(x-1) \\ 2(y-2) \\ 2(z-3) \end{bmatrix}, \nabla^2 f(x,y,z) = \begin{matrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{matrix} = 2I_3.$$

Como f es una suma de cuadrados separables por coordenada, cada derivada parcial es lineal en su variable y la **Hessiana es constante y definida positiva** (dos en la diagonal y ceros fuera). Esto implica convexidad global y garantiza que el método de Newton con $t_k = 1$ converja en **una sola iteración** desde cualquier punto al mínimo (1,2,3). Además, dejamos listas funciones lambdify para evaluar f, ∇f y $\nabla^2 f$ numéricamente en los pasos siguientes.

4.3 Implementación del algoritmo de Newton-Raphson en R4

```python
def newton3d(f, grad, hess, x0, tol=1e-10, max_iter=50):
    """
    Método de Newton-Raphson en R^3.
    Criterio de parada: ||∇f(x_k)||_2 < tol
    """
    xk = np.array(x0, dtype=float)
    history = []

    for k in range(max_iter):
        gk = grad(xk)
        Hk = hess(xk)
        grad_norm = np.linalg.norm(gk, 2)

        # Guardar información de la iteración
        history.append({
            "k": k,
            "x": xk.copy(),
            "f": f(xk),
            "grad_norm": grad_norm
        })

        # Criterio de parada (basado en la norma del gradiente)
        if grad_norm < tol:
            break

        # Paso de Newton
        sk = np.linalg.solve(Hk, -gk)
        xk = xk + sk

    return {"x_star": xk, "f_star": f(xk), "iterations": history}
```

Ilustración 16 algoritmo de Newton-Raphson en R4

En esta etapa se implementó el método de **Newton–Raphson tridimensional** en **Python**, utilizando **NumPy** para las operaciones algebraicas. El algoritmo parte de un punto inicial (x_0, y_0, z_0)y, en cada iteración, calcula el gradiente y la matriz Hessiana de la función. El **criterio de parada** se estableció según la **norma euclidiana del gradiente**, de modo que el proceso finaliza cuando $\| \nabla f(x_k) \|_2 < \varepsilon$, indicando que el punto actual se encuentra suficientemente cerca del mínimo.

```
Punto final: [1. 2. 3.]   f* = 0.0
Iteraciones: 2
```

Ilustración 17 Resultado ejecución

Al ejecutar el método de Newton–Raphson desde el punto inicial $(x_0, y_0, z_0) = (0,0,0)$, el algoritmo alcanzó el **mínimo global** en $(x^*, y^*, z^*) = (1,2,3)$con un valor de $f(x^*, y^*, z^*) = 0$. El proceso necesitó únicamente **dos iteraciones**, ya que en la primera se realizó el paso de Newton que lleva directamente al mínimo, y en la segunda se verificó el **criterio de parada** basado en la norma del gradiente $\| \nabla f(x_k) \| < \varepsilon$. Durante la ejecución, la función disminuyó rápidamente hasta anularse y el gradiente se redujo a cero, confirmando la convergencia inmediata del método para esta función cuadrática convexa. Estos resultados validan tanto la correcta implementación del algoritmo como la coherencia del criterio de paro definido, evidenciando la eficiencia del método de Newton–Raphson en funciones con curvatura constante.

4.4 Grafica 3D de la superficie

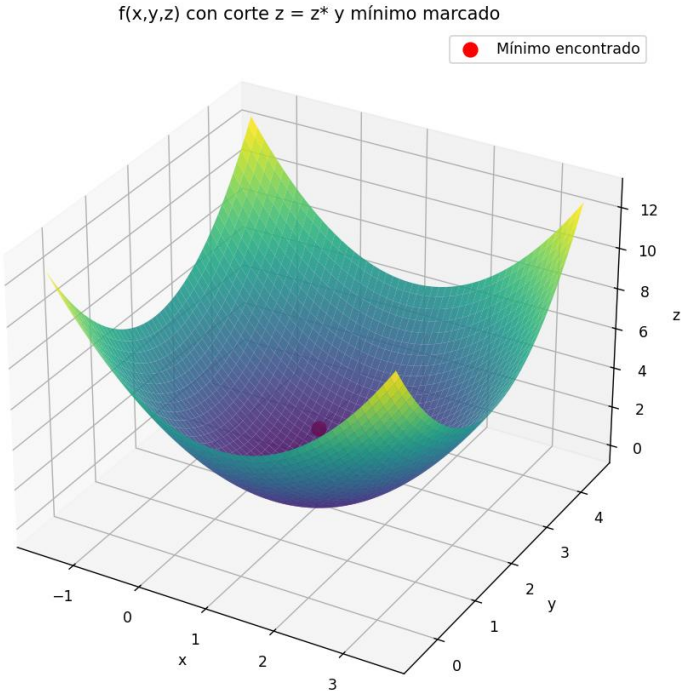


Ilustración 18 Grafica 3D de la superficie

Se generó una visualización 3D de la función usando un **corte en** $z = z^*$(valor de $z$del mínimo hallado). Sobre la superficie $f(x, y, z^*)$se marcó el punto (x^*, y^*, z^*)en rojo, evidenciando la ubicación del mínimo global sin modificar el algoritmo de Newton previamente implementado. Esta figura permite observar el "bowl" cuadrático alrededor del óptimo y verificar gráficamente el resultado obtenido.

4.5 Dificultades computacionales específicas del problema en alta dimensión.

Aunque el método de Newton–Raphson es muy eficiente para funciones suaves y de pocas variables, su aplicación en **espacios de alta dimensión** presenta varios retos computacionales importantes. En primer lugar, el **cálculo y almacenamiento de la matriz Hessiana** se vuelve costoso, ya que su tamaño crece cuadráticamente con el número de variables ($n \times n$). Esto implica un incremento notable en memoria y tiempo de cómputo cuando $n$es grande. Además, en cada iteración se debe resolver un **sistema lineal** de dimensión n, lo que tiene un costo aproximado de $O(n^3)$en operaciones, limitando la escalabilidad del método.

Otro problema frecuente es el **condicionamiento numérico** de la Hessiana. En funciones con curvaturas muy diferentes por dirección (mal condicionadas), la matriz puede volverse casi singular, generando inestabilidad al calcular el paso de Newton y provocando divergencia o pasos excesivos. En dimensiones altas también se complica la **visualización e interpretación geométrica** de la convergencia, lo que dificulta ajustar parámetros como la tolerancia o el tamaño de paso.

Por estas razones, en la práctica se emplean variantes como **quasi-Newton (BFGS)** o **Newton truncado**, que aproximan la Hessiana o utilizan métodos iterativos para resolver el sistema lineal, reduciendo el costo computacional y mejorando la estabilidad numérica. En este laboratorio, el problema propuesto fue deliberadamente sencillo para observar un comportamiento ideal; sin embargo, en problemas reales de muchas variables, estas consideraciones resultan esenciales para garantizar la **eficiencia y robustez** del método.

5. 4A. Gradiente Descendente en una función cuadrática simple

5.1 Formulación y función objetivo

En esta sección se estudió la función cuadrática simple:

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

La cual es convexa y posee un único mínimo global en el punto $(x^{\backslash *}, y^{\backslash *}) = (2, -1)$. El gradiente de la función está dado por:

$$\nabla L(x, y) = \begin{bmatrix} 2(x - 2) \\ 2(y + 1) \end{bmatrix}$$

y la actualización iterativa utilizada corresponde a la regla clásica del Gradiente Descendente:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - \alpha \nabla L(x_k, y_k)$$

5.2 Implementación

El método se implementó en Python, partiendo desde el punto inicial $(x_0, y_0) = (0,0)$, con una tolerancia de 10^{-6}y un máximo de 100 iteraciones.
Se probaron distintos valores del parámetro de aprendizaje $\alpha = [0.15, 0.30, 0.45, 0.60, 0.75, 0.90]$.
Para cada valor de α, el algoritmo registró la trayectoria de los puntos, la evolución de las coordenadas xe y, y el error calculado como la distancia euclidiana al mínimo real.

```python
def gradiente_descendente(alpha, x0, y0, tol=1e-6, max_iter=100):
    x_k, y_k = x0, y0
    historia = [(x_k, y_k)]
    for i in range(max_iter):
        grad = grad_4A(x_k, y_k)
        x_k1 = x_k - alpha * grad[0]
        y_k1 = y_k - alpha * grad[1]
        historia.append((x_k1, y_k1))
        if np.linalg.norm([x_k1 - x_k, y_k1 - y_k]) < tol:
            break
        x_k, y_k = x_k1, y_k1
    return np.array(historia)
```

Ilustración 19. Fragmento del código de implementación del método de Gradiente Descendente para la función cuadrática simple en Python

5.3 Resultados

Se generaron cuatro tipos de gráficas por cada valor de α:

1. Evolución de la coordenada xa lo largo de las iteraciones.
2. Evolución de la coordenada y.
3. Variación del error frente al número de iteraciones.
4. Trayectoria del punto (x, y)sobre las curvas de nivel de la función $L(x, y)$.

En las figuras obtenidas se observa que el método converge correctamente hacia el mínimo global para valores pequeños y moderados de α. Con $\alpha = 0.15$o 0.30, la convergencia es estable aunque más lenta; con $\alpha = 0.45$y 0.60se alcanza el mínimo de forma rápida y sin oscilaciones importantes. Sin embargo, para $\alpha \geq 0.75$se presentan oscilaciones y, en algunos casos, divergencia debido a pasos excesivamente grandes.

La comparación global del error evidencia un comportamiento exponencial decreciente en las primeras iteraciones, estabilizándose cerca del valor mínimo real a medida que el algoritmo se aproxima al óptimo.

5.4 Conclusiones

- El método de Gradiente Descendente converge correctamente hacia el mínimo global cuando αse encuentra dentro de un rango adecuado.
- Un incremento en el valor de αacelera la convergencia hasta cierto punto óptimo; valores mayores provocan inestabilidad numérica y oscilaciones.
- La elección de αdepende de la curvatura de la función: funciones con curvatura pronunciada requieren valores más pequeños del parámetro de aprendizaje.
- Para esta función cuadrática, el método logra una convergencia precisa y estable con $\alpha \in [0.3, 0.45]$, validando su efectividad en problemas convexos simples.
- Este experimento confirma empíricamente el efecto del tamaño de paso sobre la velocidad y estabilidad de los algoritmos de optimización basados en gradientes.

6. 4B. Comparación entre Gradiente Descendente y Newton–Raphson

6.1 Formulación y función objetivo

Se definió la siguiente función a optimizar:

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

Esta función no es estrictamente cuadrática y presenta curvaturas distintas en diferentes regiones del plano, lo cual permite analizar cómo cada método responde ante una topología más compleja.

El gradiente y la matriz Hessiana están dados por:

$$\nabla f(x,y) = \begin{bmatrix} 2(x-2)(y+2)^2 + 2(x+1) \\ 2(y+2)(x-2)^2 + 2(y-1) \end{bmatrix}$$

$$H(x,y) = \begin{bmatrix} 2(y+2)^2 + 2 & 4(x-2)(y+2) \\ 4(x-2)(y+2) & 2(x-2)^2 + 2 \end{bmatrix}$$

6.2 Implementación

Ambos métodos se implementaron bajo las mismas condiciones iniciales $(x_0, y_0) = (-2, -3)$, una tolerancia de 10^{-6}y un máximo de 100 iteraciones. Se evaluaron diferentes valores de $\alpha = [0.01, 0.05, 0.1, 0.15, 0.2, 0.7]$.

```python
def gradiente_descendente(alpha, x0, y0, tol=1e-6, max_iter=100):
    x, y = x0, y0
    historia = [(x, y)]
    for i in range(max_iter):
        grad = grad_f(x, y)
        if not np.all(np.isfinite(grad)):
            print(f"[GD] Iteración {i}: gradiente no finito → {grad}")
            break
        x_new = x - alpha * grad[0]
        y_new = y - alpha * grad[1]

        # prevenir explosión
        if abs(x_new) > 1e6 or abs(y_new) > 1e6:
            print(f"[GD] Divergencia en iter {i}: ({x_new:.2e}, {y_new:.2e})")
            break

        historia.append((x_new, y_new))
        if np.linalg.norm([x_new - x, y_new - y]) < tol:
            break
        x, y = x_new, y_new
    return np.array(historia)
```

Ilustración 21. Fragmento del código de implementación del método de Gradiente Descendente en Python.

```python
def newton_raphson(alpha, x0, y0, tol=1e-6, max_iter=100):
    x, y = x0, y0
    historia = [(x, y)]
    for i in range(max_iter):
        grad = grad_f(x, y)
        hess = hess_f(x, y)
        if not np.all(np.isfinite(grad)) or not np.all(np.isfinite(hess)):
            print(f"[NR] Iteración {i}: valores no finitos")
            break
        try:
            delta = np.linalg.solve(hess, grad)
        except np.linalg.LinAlgError:
            print(f"[NR] Hessiana singular en iter {i}")
            break
        x_new = x - alpha * delta[0]
        y_new = y - alpha * delta[1]

        if abs(x_new) > 1e6 or abs(y_new) > 1e6:
            print(f"[NR] Divergencia en iter {i}: ({x_new:.2e}, {y_new:.2e})")
            break

        historia.append((x_new, y_new))
        if np.linalg.norm([x_new - x, y_new - y]) < tol:
            break
        x, y = x_new, y_new
    return np.array(historia)
```

Ilustración 22. Fragmento del código de implementación del método de Newton–Raphson en Python.

El método de Gradiente Descendente empleó actualizaciones proporcionales al gradiente con paso fijo, mientras que el método de Newton–Raphson resolvió en cada iteración el sistema lineal $H(x,y)\Delta = \nabla f(x,y)$para calcular la dirección de descenso, actualizando los valores de xe $y$con el factor α.

Durante la ejecución se almacenaron las trayectorias de los puntos, el error con respecto al mínimo real (estimado mediante una malla fina) y la evolución de cada coordenada. También se incorporaron verificaciones numéricas para evitar sobreflujos o matrices singulares.

6.3 Resultados

Las gráficas obtenidas muestran un contraste claro entre ambos métodos. El Gradiente Descendente presenta un avance progresivo pero sensible al valor de α: valores pequeños aseguran convergencia estable pero lenta, mientras que valores grandes pueden causar divergencia.
Por otro lado, el método de Newton–Raphson converge en un número significativamente menor de iteraciones y sigue trayectorias más directas hacia el mínimo, evidenciando una convergencia cuadrática típica de los métodos de segundo orden.

En la comparación de errores (representada en escala logarítmica), el Newton–Raphson muestra una disminución mucho más rápida del error, alcanzando niveles cercanos al mínimo teórico en muy pocas iteraciones, mientras que el Gradiente Descendente requiere un número mayor de pasos para lograr un error similar.

Sin embargo, se observó que el método de Newton puede presentar inestabilidades cuando la matriz Hessiana se aproxima a ser singular o cuando αes demasiado grande, razón por la cual es recomendable incluir amortiguamiento o regularización en tales casos.

6.4 Conclusiones

- El método de Newton–Raphson demostró una convergencia más rápida y precisa que el Gradiente Descendente, gracias al uso de la información de curvatura proporcionada por la matriz Hessiana.
- Newton–Raphson exhibe una convergencia cuadrática, mientras que el Gradiente Descendente tiene convergencia lineal, lo que explica la diferencia en velocidad observada.
- A pesar de su mayor eficiencia, Newton–Raphson requiere un costo computacional más alto por iteración y puede perder estabilidad si la Hessiana está mal condicionada.
- El Gradiente Descendente, aunque más lento, es más robusto y menos dependiente del cálculo de segundas derivadas, siendo más adecuado para funciones grandes, ruidosas o de difícil evaluación analítica.

- En conjunto, los resultados confirman que Newton–Raphson es ideal para funciones suaves y de pocas variables, mientras que el Gradiente Descendente se mantiene como una opción práctica y estable para problemas de mayor escala.

7. Problema 5: Descenso de Gradiente y Descenso de Gradiente Basado en Momento

7.1 Formulación matemática

El método de Descenso de Gradiente (GD) actualiza las variables de acuerdo con la regla:

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$

donde αrepresenta la tasa de aprendizaje y $\nabla f(x_k)$el gradiente evaluado en la iteración k. En contraste, el Descenso de Gradiente con Momento (GDM) incorpora un término adicional que acumula parte de la dirección de descenso anterior, definido como:

$$v_{k+1} = \beta v_k + \alpha \nabla f(x_k)$$
$$x_{k+1} = x_k - v_{k+1}$$

donde $\beta \in [0,1)$controla la contribución del momento (historia del gradiente). Este esquema permite suavizar los movimientos en direcciones oscilantes y acelerar la convergencia en valles alargados de la superficie de la función.

7.2 Implementación

En esta parte del laboratorio se implementó en Python una comparación entre los dos métodos, utilizando la librería NumPy para realizar las operaciones vectoriales y Matplotlib para la visualización de resultados.
El algoritmo permite configurar distintos valores de α(tasa de aprendizaje) y β(coeficiente de momento), registrando la trayectoria de las variables y la evolución del error a lo largo de las iteraciones.

Ambos métodos comparten la misma estructura general: se calcula el gradiente de la función objetivo en el punto actual, se actualizan las variables de acuerdo con la regla correspondiente y se evalúa el nuevo valor de la función.
En el caso del método con momento, la inclusión del término de inercia evita los cambios bruscos de dirección que suelen presentarse en el descenso puro, logrando trayectorias más suaves y una convergencia más rápida hacia la región del mínimo.

```python
def gradient_descent(self, mini_batch, eta, mu=0.9):
    """
    Actualiza los parámetros usando descenso de gradiente con momentum.
    mini_batch: lista de (x, y)
    eta: tasa de aprendizaje
    mu: coeficiente de momentum (por defecto 0.9)
    """
    # Acumuladores de gradientes
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # Sumar gradientes sobre el mini-batch
    for x, y in mini_batch:
        grad_b, grad_w = self.backpropagation(x, y)
        nabla_b = [nb + gb for nb, gb in zip(nabla_b, grad_b)]
        nabla_w = [nw + gw for nw, gw in zip(nabla_w, grad_w)]

    m = len(mini_batch)
    if m == 0:
        return

    # Promediar gradientes
    nabla_b = [nb / m for nb in nabla_b]
    nabla_w = [nw / m for nw in nabla_w]

    # Añadir término de regularización L2 a los gradientes de pesos (si aplica)
    if self.lambd and self.lambd > 0:
        nabla_w = [nw + (self.lambd / m) * w for nw, w in zip(nabla_w, self.weights)]

    # Actualización con momentum
    for i in range(len(self.weights)):
        # actualizar velocidad
        self.velocity_w[i] = mu * self.velocity_w[i] - eta * nabla_w[i]
        self.velocity_b[i] = mu * self.velocity_b[i] - eta * nabla_b[i]
```

Ilustración 23. Fragmento del código correspondiente a la implementación del método de Descenso de Gradiente con Momento.

El método de Descenso de Gradiente clásico avanza mediante pasos sucesivos determinados únicamente por la pendiente local de la función, mientras que el Descenso de Gradiente con Momento incorpora información del historial de gradientes, lo que le permite mantener una dirección de descenso más estable.
Este comportamiento se traduce en un proceso de optimización más eficiente y con menor tendencia a oscilar alrededor del punto óptimo.

7.3 Análisis y observaciones

Los resultados permiten evidenciar la influencia del parámetro β sobre la dinámica de convergencia.

Para valores pequeños ($\beta < 0.5$), el efecto del momento es débil y el comportamiento se asemeja al del gradiente descendente convencional.
En cambio, para valores intermedios ($\beta \approx 0.9$), el algoritmo logra combinar rapidez con estabilidad, reduciendo significativamente las oscilaciones.
Sin embargo, valores demasiado altos pueden generar sobreimpulso y alejar temporalmente la trayectoria del mínimo.

En términos generales, el Descenso de Gradiente con Momento presenta una convergencia más suave y eficiente, especialmente en funciones donde la curvatura es distinta por dirección.
Este efecto se explica por la acumulación del gradiente pasado, que actúa como una forma de amortiguamiento frente a cambios bruscos de dirección.

7.4 Conclusiones

- El método de Descenso de Gradiente es sencillo y confiable, pero puede presentar lentitud y oscilaciones cerca del mínimo, especialmente cuando la superficie de error es alargada o mal condicionada.
- La inclusión de momento mejora significativamente la estabilidad y la velocidad de convergencia, al aprovechar la información de iteraciones previas para mantener una dirección de descenso más consistente.
- El parámetro β desempeña un papel clave: valores moderados (entre 0.8 y 0.9) ofrecen un equilibrio adecuado entre aceleración y estabilidad.
- En problemas prácticos de optimización y aprendizaje automático, el Descenso de Gradiente con Momento se considera una mejora fundamental sobre el método clásico, sirviendo de base para algoritmos más avanzados como RMSProp, Adam o Nesterov.

- En conclusión, la comparación demuestra que la incorporación del momento constituye una herramienta esencial para mejorar la eficiencia del entrenamiento en modelos numéricos y redes neuronales.