Universidad de La Habana
Facultad de Matemática y Computación

Proyecto de Modelos de Optimización - Función de Rosenbrock modificada

Ariel González Gómez (C-312)

6 de septiembre de 2024

1. Resumen

Este trabajo realiza un análisis exhaustivo de diferentes algoritmos de optimización aplicados a la función de Rosenbrock modificada, una función no lineal diseñada para desafiar la capacidad de los algoritmos de encontrar su mínimo global debido a la presencia de múltiples mínimos locales. Se implementaron tanto métodos clásicos como algoritmos evolutivos, poblacionales y de búsqueda estocástica. Los métodos clásicos incluyen el descenso por gradiente, Broyden–Fletcher–Goldfarb–Shanno (BFGS), el método de región de confianza y Nelder-Mead, que exploran la función utilizando gradientes pero enfrentan dificultades para evitar los mínimos locales. Por otro lado, los algoritmos evolutivos, como la evolución diferencial y el algoritmo genético, proporcionan una exploración más robusta del espacio de búsqueda mediante mecanismos de selección, mutación y cruzamiento, permitiendo escapar de estos mínimos locales. Los algoritmos basados en población, como la optimización por enjambre de partículas (PSO) y la estrategia de adaptación de la matriz de covarianza (CMA-ES), colaboran entre múltiples soluciones potenciales para mejorar la convergencia. Finalmente, los métodos de búsqueda estocástica, como el recocido simulado y el salto de cuenca, permiten explorar el espacio de soluciones aceptando temporalmente soluciones subóptimas, lo que les otorga una ventaja para encontrar el mínimo global en funciones multimodales. La comparación entre estos algoritmos se realiza considerando tanto su capacidad para hallar soluciones óptimas como su eficiencia computacional. Repositorio en GitHub del proyecto: https://github.com/arielgg46/Optimization-Models-Project.

2. Introducción

La **función de Rosenbrock** es una de las funciones de referencia más comunes en la optimización numérica. Fue diseñada para probar la eficacia de los algoritmos de optimización en problemas no lineales. También se le conoce como la "función del valle" porque, aunque tiene un mínimo global bien definido, este está ubicado dentro de un valle largo, curvo y estrecho, lo que dificulta que los algoritmos de optimización estándar lo encuentren rápidamente.

La función de Rosenbrock original está definida para n dimensiones y se caracteriza por tener un único mínimo global en $x^* = (1, 1, \ldots, 1)$, pero un gradiente suave que puede confundir a muchos algoritmos de búsqueda de mínimos, lo que la convierte en una prueba exigente.

La función de Rosenbrock en n dimensiones está definida como:

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[(a - x_i)^2 + b \cdot (x_{i+1} - x_i^2)^2 \right]$$

donde $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ es el vector de variables y a y b son parámetros.

Para el caso clásico con $a = 1$ y $b = 100$, (imagen 1) la función se convierte en:

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[(1 - x_i)^2 + 100 \cdot (x_{i+1} - x_i^2)^2\right]$$

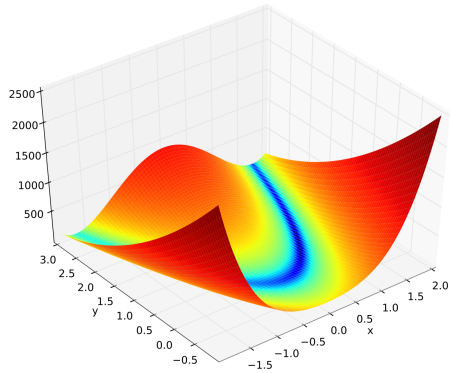

Figura 1: La trama de la Rosenbrock en función de dos variables. Aquí $a = 1, b = 100$ y el valor mínimo es cero en $(1, 1)$.

En el presente trabajo se hace un análisis de una modificación de la función de Rosenbrock:

$$f(x) = 74 + 100 \cdot (x_2 + x_1^2)^2 + (1 - x_1^2) - 400 \cdot e^{-\frac{(x_1+1)^2+(x_2+1)^2}{0,1}} \tag{1}$$

sujeta a $-2 \leq x_1, x_2 \leq 2$.

En esta **modificación** de la función de Rosenbrock, se ha añadido un término exponencial que genera un "bulto" gaussiano en la región alrededor del punto $(-1, -1)$. Esto introduce un varios mínimos locales, mientras que el mínimo global sigue estando alrededor de $(-1, -1)$. La adición de este bulto hace que la función sea más difícil de optimizar porque los algoritmos pueden quedar atrapados en los mínimos locales (ver imagen 2).

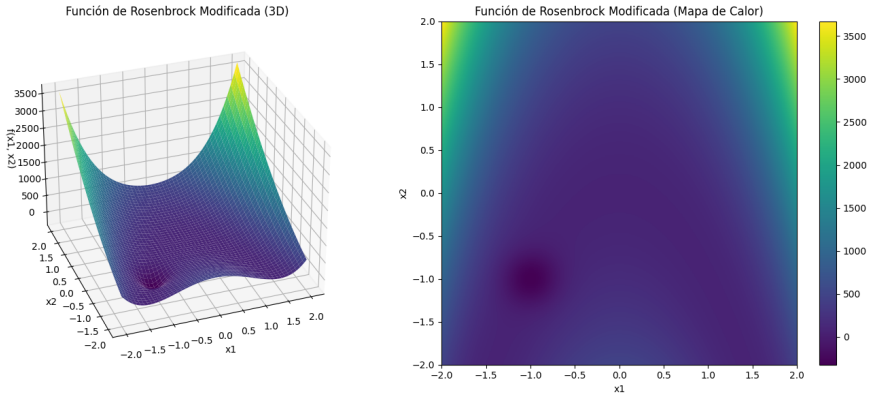

Figura 2: Función de Rosenbrock modificada

3. Preliminares

La función modificada de Rosenbrock presenta algunas características que influyen en la búsqueda de sus puntos mínimos:

- **Continua**: La función es continua en el dominio $[-2, 2]$ para ambas variables x_1 y x_2. Esto significa que no presenta saltos ni discontinuidades, lo que asegura que cualquier algoritmo de optimización puede navegar por el espacio de búsqueda de manera fluida. La continuidad es esencial para que los algoritmos basados en gradientes y otros métodos numéricos puedan aproximarse progresivamente a un mínimo sin interrupciones.

- **Diferenciable**: La función es completamente diferenciable, lo que implica que su gradiente está bien definido en todo su dominio. La diferenciabilidad es crucial para los algoritmos que dependen del cálculo del gradiente o de la derivada, como el método de gradiente descendente o Newton-Raphson. Esta propiedad permite a los algoritmos usar la información direccional proporcionada por las derivadas para encontrar el mínimo más rápidamente.

- **No Separabilidad**: La no separabilidad de la función indica que las variables x_1 y x_2 están entrelazadas, es decir, no es posible dividir la función en subproblemas independientes para cada variable. Esto implica que los algoritmos deben optimizar simultáneamente todas las variables involucradas. En la práctica, los problemas no separables suelen ser más complejos y requieren algoritmos capaces de manejar interacciones entre las variables, como los algoritmos genéticos o enfriamiento simulado.

- **No Escalable**: La función, en su forma actual, está diseñada específicamente para un problema de dos dimensiones, y la estructura no permite una escalabilidad directa a dimensiones superiores. Esto limita su aplicabilidad a problemas de mayor dimensión sin una modificación significativa. La falta de escalabilidad es una consideración importante cuando se eligen algoritmos, ya que algunos algoritmos de optimización, como PSO o DE, son más efectivos en problemas de alta dimensión, mientras que otros pueden no ser tan eficientes si no pueden adaptarse bien a problemas de múltiples dimensiones.

- **Multimodal**: La multimodalidad de la función se manifiesta por la presencia de múltiples mínimos (locales y globales), donde los algoritmos pueden quedarse "atrapados" en los mínimos locales. Este fenómeno introduce desafíos adicionales, ya que los algoritmos simples, como el gradiente descendente, pueden converger a mínimos locales en lugar de alcanzar el mínimo global. Por lo tanto, es recomendable utilizar algoritmos de optimización global, como algoritmos evolutivos, recocido simulado o búsqueda por enjambre de partículas (PSO), que están diseñados para explorar eficientemente el espacio de búsqueda y evitar caer en mínimos locales.

En resumen, la elección del algoritmo de optimización dependerá de la capacidad del algoritmo para aprovechar las propiedades diferenciales y continuas de la función, mientras aborda los desafíos de la no separabilidad y la multimodalidad. En el presente trabajo se utilizaron varias clases de algoritmos para la optimización de la función: métodos clásicos, algoritmos evolutivos, algoritmos de optimización basados en población, y algoritmos de búsqueda estocástica. Cada uno tiene sus ventajas y desventajas, y se comportan de manera distinta al aplicarlos sobre la función de Rosenbrock modificada.

¡Se implementaron un gran total de 10 algoritmos! Afortunadamente el todopoderoso Python en su infinita sabiduría[1] tiene implementaciones de la mayoría de estos métodos.

4. Algoritmos de Optimización

4.1. Métodos Clásicos

Los métodos clásicos de optimización son enfoques basados en gradientes y derivadas que buscan puntos óptimos de manera sistemática. Debido a que la función modificada de Rosenbrock es diferenciable, estos métodos pueden aprovechar la información de las derivadas. Sin embargo, debido a su naturaleza multimodal, estos métodos podrían quedar atrapados en mínimos locales. Se espera que los métodos como el descenso máximo o cuasi-Newton tengan dificultades en encontrar el mínimo global en $(-1, -1)$, a menos que se inicie cerca de la solución global.

[1] Bibliotecas tramposas.

4.1.1. Cálculo de gradientes

Para la aplicación de la mayoría de los métodos clásicos utilizados se necesita el cálculo del gradiente de la función a optimizar. Este se obtiene hallando las derivadas parciales de la función con respecto a los variables que presenta (x_1 y x_2).

El gradiente de $f(x)$ es:

$$\frac{\partial f}{\partial x_1} = 400 \cdot x_1 \cdot (x_1^2 + x_2) - 2 \cdot x_1 + 8000 \cdot (x_1 + 1) \cdot e^{-10 \cdot (x_1+1)^2 - 10 \cdot (x_2+1)^2}$$

$$\frac{\partial f}{\partial x_2} = 200 \cdot x_1^2 + 200 \cdot x_2 + 8000 \cdot (x_2 + 1) \cdot e^{-10 \cdot (x_1+1)^2 - 10 \cdot (x_2+1)^2}$$

Implementación en Python:

```python
# Funcion de gradiente de la funcion modificada de Rosenbrock
def rosenbrock_modified_gradient(x):
    x1, x2 = x
    grad_x1 = 400*x1*(x1**2+x2) - 2*x1 + 8000*(x1+1)*np.exp(-10*(x1+1)**2 - 10*(x2+1)**2)
    grad_x2 = 200*x1**2 + 200*x2 + 8000*(x2+1)*np.exp(-10*(x1+1)**2 - 10*(x2+1)**2)
    return np.array([grad_x1, grad_x2])
```

4.1.2. Descenso Máximo (Gradient Descent - GD)

El descenso por gradiente es uno de los algoritmos más básicos de optimización. Su objetivo es minimizar una función moviéndose iterativamente en la dirección opuesta al gradiente, que indica la dirección de la mayor tasa de incremento de la función. Dado un punto inicial, el algoritmo actualiza la solución en la forma:

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

donde α es la tasa de aprendizaje, y $\nabla f(x_k)$ es el gradiente de la función en el punto x_k. Un aspecto clave de este algoritmo es la selección adecuada de α. Si es demasiado grande, el algoritmo puede rebotar alrededor del mínimo sin converger; si es demasiado pequeño, puede requerir muchas iteraciones para alcanzar el mínimo, lo que lo hace ineficiente.

Teóricamente, el descenso por gradiente garantiza la convergencia en funciones convexas diferenciables, pero su desempeño en funciones no convexas, como la modificada de Rosenbrock, es más complicado. Al tratarse de una función multimodal, el algoritmo puede quedar atrapado fácilmente en mínimos locales, a menos que se inicie cerca del mínimo global o se utilicen estrategias avanzadas como el ajuste dinámico del tamaño del paso. Al ser continua y diferenciable, la función permite aplicar este algoritmo, pero su estructura no separable puede hacer que los gradientes en algunas direcciones dominen, dificultando la búsqueda del mínimo en otras.

- **Ventajas:**
  - Sencillo de implementar y computacionalmente barato.
- **Desventajas:**
  - Puede quedar atrapado en mínimos locales.
  - Sensible a la elección del tamaño del paso (learning rate).

Implementación:

```python
def gradient_descent(grad_func, x0, bounds, lr=0.001, max_iter=1000):
    x = x0
    for i in range(max_iter):
        grad = grad_func(x)
        x = x - lr * grad
        # Limitar x dentro de los bordes especificados
        x = np.clip(x, bounds[0][0], bounds[0][1])
        x = np.clip(x, bounds[1][0], bounds[1][1])
```

```
        if np.linalg.norm(grad) < 1e-6:
            break
    return x

def run_gradient_descent(x0):
    result = gradient_descent(rosenbrock_modified_gradient, x0, bounds=bounds)
    return result
```

Resultado:
Punto inicial: $x_0 = (0, 0)$
Solución hallada: $x^* = (-0{,}000052565750173, -0{,}000000085232503)$, $f(x^*) = 74{,}999999171907461$
Tiempo de ejecución: 0.053996 segundos

4.1.3. Algoritmo de Broyden–Fletcher–Goldfarb–Shanno - BFGS

El BFGS es un algoritmo cuasi-Newton que mejora la eficiencia del descenso por gradiente aproximando la matriz Hessiana inversa, que contiene información sobre la curvatura de la función. A diferencia del descenso por gradiente, que solo utiliza información de la pendiente, BFGS utiliza la curvatura para ajustar la dirección y el tamaño del paso de manera más precisa, acelerando la convergencia.

La actualización de la matriz Hessiana inversa se realiza en cada iteración mediante la siguiente fórmula de actualización de BFGS:

$$H_{k+1} = H_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{H_k s_k s_k^T H_k}{s_k^T H_k s_k}$$

donde $s_k = x_{k+1} - x_k$ y $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$. Esta aproximación no requiere el cálculo directo de la Hessiana, lo que la hace más eficiente en problemas donde calcular segundas derivadas es computacionalmente costoso.

Al ser un método que depende de gradientes y de aproximaciones de la curvatura, BFGS es muy efectivo en funciones diferenciables como la de Rosenbrock modificada. Sin embargo, en funciones multimodales, este método puede también quedar atrapado en mínimos locales. Además, como BFGS se basa en la curvatura local, su rendimiento puede degradarse si la matriz Hessiana es mal condicionada, lo cual puede ocurrir en el valle estrecho de la función de Rosenbrock. BFGS es adecuado para funciones de baja dimensión, pero puede volverse costoso en problemas no escalables o con muchas variables, debido al tamaño de la matriz Hessiana.

- **Ventajas:**
  - Convergencia rápida en comparación con el descenso máximo.
  - No requiere cálculo explícito de la Hessiana.
- **Desventajas:**
  - Puede ser costoso en términos de memoria y computación en problemas de alta dimensionalidad.
  - Posibilidad de quedarse en mínimos locales.

Implementación:

```
from scipy.optimize import minimize

def run_bfgs(x0):
    result = minimize(rosenbrock_modified, x0, method='L-BFGS-B', jac=
    rosenbrock_modified_gradient, bounds=bounds)
    return result.x
```

Resultado:
Punto inicial: $x_0 = (0, 0)$
Solución hallada: $x^* = (-1{,}000227822852636, -1{,}000011114532771)$, $f(x^*) = -326{,}000227825111438$
Tiempo de ejecución: 0.035000 segundos

4.1.4. Método de Región de Confianza

El método de región de confianza es una estrategia de optimización que ajusta un modelo cuadrático de la función objetivo dentro de una región alrededor del punto actual. A diferencia de métodos de búsqueda lineal como el descenso por gradiente, que ajusta la dirección de búsqueda globalmente, el método de región de confianza evalúa la calidad del modelo local y ajusta el tamaño de la región de búsqueda.

En cada iteración, el algoritmo resuelve el siguiente problema dentro de la región de confianza:

$$\min_m \; q_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p$$

sujeto a la restricción $\|p\| \leq \Delta_k$, donde Δ_k es el radio de la región de confianza, p es el desplazamiento propuesto, y B_k es una aproximación de la matriz Hessiana. Si el modelo es una buena representación de la función dentro de la región, se expande el radio Δ_k; si no, se reduce.

Este enfoque es más robusto que el descenso por gradiente o BFGS en problemas con comportamiento no lineal fuerte o gradientes mal condicionados, como en el caso de la función de Rosenbrock modificada, donde la función presenta un valle estrecho que desafía a los métodos estándar. El método de región de confianza es ideal para funciones no separables, ya que puede ajustar localmente su aproximación. Sin embargo, su costo computacional es mayor, especialmente si la función no es escalable, porque cada iteración requiere la resolución de un subproblema cuadrático.

- **Ventajas:**
  - Capaz de manejar funciones difíciles, especialmente cuando las aproximaciones son buenas.
  - Robustez frente a problemas de mal comportamiento en el gradiente.
- **Desventajas:**
  - Complejidad adicional en la implementación y ajuste de parámetros de región de confianza.
  - Computacionalmente costoso.

Implementación:

```
from scipy.optimize import minimize

def run_trust_region(x0):
    result = minimize(rosenbrock_modified, x0, method='trust-constr', jac=
    rosenbrock_modified_gradient, bounds=bounds)
    return result.x
```

Resultado:
Punto inicial: $x_0 = (0, 0)$
Solución hallada: $x^* = (-1{,}000227823666112, -1{,}000011114940307)$, $f(x^*) = -326{,}000227825111438$
Tiempo de ejecución: 1.037826 segundos

4.1.5. Método Nelder-Mead (*Simplex*)

El método Nelder-Mead es un algoritmo de optimización sin derivadas que utiliza un enfoque geométrico. Consiste en evaluar la función en los vértices de un simplex (un poliedro de $n + 1$ puntos en un espacio de dimensión n) y ajustar iterativamente la forma del simplex para encontrar el mínimo. Este método es particularmente útil cuando las derivadas de la función no están disponibles o son difíciles de calcular.

El algoritmo realiza operaciones como expansión, contracción y reducción del simplex, basándose en las evaluaciones de la función objetivo en los vértices. Aunque es simple de implementar y no requiere información de gradientes, su principal debilidad es que no garantiza la convergencia al mínimo global y puede quedar atrapado en mínimos locales, especialmente en funciones multimodales como la de Rosenbrock modificada.

En términos teóricos, Nelder-Mead no asume que la función sea diferenciable, lo que lo hace útil en casos donde los métodos basados en derivadas fallan. Sin embargo, como la función de Rosenbrock modificada es continua y diferenciable, este método podría no aprovechar todas las ventajas de la estructura de la función. Dado que la función es no escalable, el método Nelder-Mead, que escala de manera exponencial con el número de variables, se vuelve ineficiente en dimensiones superiores, limitando su aplicabilidad.

- **Ventajas:**
  - No requiere derivadas. Ideal para problemas donde la derivada de la función no está disponible o es costosa de calcular.
  - Fácil de implementar y entender.
  - Buena para problemas de optimización no lineales. Funciona bien en espacios de alta dimensión y con funciones no suaves.
- **Desventajas:**
  - No garantiza la convergencia global. Puede quedar atrapado en óptimos locales y no siempre encontrará el mínimo global.
  - Dependencia del punto inicial. El resultado puede ser sensible al punto inicial elegido.
  - Rendimiento en problemas de alta dimensión. Puede ser menos eficiente en problemas con muchas dimensiones en comparación con otros métodos de optimización.

Implementación:

```python
from scipy.optimize import minimize

# Definicion de la funcion modificada de Rosenbrock con penalizacion para manejar restricciones
def rosenbrock_modified_penalized(x):
    x1, x2 = x
    # Penalizacion por estar fuera de los limites
    penalty = 0
    if x1 < bounds[0][0] or x1 > bounds[0][1] or x2 < bounds[1][0] or x2 > bounds[1][1]:
        penalty = 1e10  # Gran penalizacion si esta fuera de los limites
    return rosenbrock_modified(x) + penalty

def run_nelder_mead(x0):
    result = minimize(rosenbrock_modified_penalized, x0=x0, method='Nelder-Mead', options={'xatol': 1e-8, 'fatol': 1e-8})
    return result.x
```

Resultado:
Punto inicial: $x_0 = (0, 0)$
Solución hallada: $x^* = (1{,}415898706924632, -1{,}999999999964812)$, $f(x^*) = 72{,}997505329285644$
Tiempo de ejecución: 0.011002 segundos

4.2. Algoritmos Evolutivos

Los algoritmos evolutivos simulan procesos de evolución biológica, utilizando operadores como mutación, cruzamiento y selección para explorar el espacio de soluciones. Estos métodos son muy efectivos en funciones multimodales, como la función modificada de Rosenbrock, ya que no dependen de derivadas y son capaces de escapar de mínimos locales.

4.2.1. Evolución Diferencial (*Differential Evolution* - DE)

La evolución diferencial es un algoritmo evolutivo que se basa en la manipulación de una población de soluciones a través de combinaciones lineales de las soluciones existentes para generar nuevas candidatas. Cada individuo en la población es representado por un vector de parámetros, y el algoritmo procede iterativamente mediante operaciones de mutación, cruce y selección.

- **Mutación**: En esta fase, se seleccionan tres soluciones aleatorias de la población y se combina la diferencia entre dos de ellas con la tercera, generando una nueva solución potencial. La fórmula general para la mutación es:

$$v_i = x_{r_1} + F \cdot (x_{r_2} - x_{r_3})$$

donde F es un factor de escala, y r_1, r_2, r_3 son índices aleatorios distintos de i.

- **Cruce**: El vector mutado se cruza con el vector actual para generar un nuevo candidato. El cruce se controla mediante una tasa de cruce CR, que decide cuántos componentes del vector mutado se transfieren al nuevo candidato.

$$u_{i,j} = \begin{cases} v_{i,j} & \text{si } rand_j(0,1) \leq CR \\ x_{i,j} & \text{de lo contrario} \end{cases}$$

- **Selección**: Después del cruce, la nueva solución generada u_i se compara con la solución original x_i, y se selecciona la mejor (la que minimice la función objetivo).

$$x_i^{(t+1)} = \begin{cases} u_i & \text{si } f(u_i) < f(x_i) \\ x_i & \text{de lo contrario} \end{cases}$$

La evolución diferencial es adecuada para la función de Rosenbrock modificada debido a su capacidad de explorar el espacio de búsqueda de manera eficaz, superando la multimodalidad y evitando los mínimos locales. Sin embargo, como la función no es escalable, el rendimiento de DE puede verse afectado si se intenta aumentar el número de dimensiones, ya que el algoritmo necesita más evaluaciones para mantener la diversidad en poblaciones de alta dimensionalidad.

- **Ventajas:**
  - Simple de implementar y efectivo para funciones continuas y multimodales.
  - Menos dependiente de la inicialización.
- **Desventajas:**
  - Puede ser más lento en la convergencia comparado con métodos basados en gradientes.

Implementación:

```python
from scipy.optimize import differential_evolution

def run_differential_evolution(x0=None):  # No necesita punto inicial
    result = differential_evolution(rosenbrock_modified, bounds)
    return result.x
```

Resultado:
Solución hallada: $x^* = (-1{,}000227828099572, -1{,}000011120362328)$, $f(x^*) = -326{,}000227825111210$
Tiempo de ejecución: 0.041525 segundos

4.2.2. Algoritmo genético (*Genetic Algorithm* - GA)

El algoritmo genético es un método inspirado en la evolución biológica, que utiliza procesos como selección, cruce y mutación para generar nuevas soluciones. La representación de cada solución se realiza mediante un cromosoma (generalmente un vector), y el proceso evolutivo busca mejorar la población a lo largo de las generaciones.

- **Selección**: Se seleccionan individuos de la población basándose en su aptitud, que se mide a través de la función objetivo. Se puede emplear diferentes estrategias de selección, como selección por torneo o ruleta, donde las soluciones con mejor aptitud tienen mayor probabilidad de ser seleccionadas.
- **Cruce**: Dos soluciones "padre" se combinan para generar descendencia. Una técnica común de cruce es el cruce de un solo punto, en el que se elige un punto de cruce al azar y los vectores de los padres se dividen y recombinan. Para un cromosoma x_1 y x_2, el cruce produce una nueva solución y:

$$y_j = \begin{cases} x_{1,j} & \text{si } j \leq \text{punto de cruce} \\ x_{2,j} & \text{de lo contrario} \end{cases}$$

- **Mutación**: Introduce diversidad en la población al alterar aleatoriamente uno o más genes en el cromosoma de una solución, evitando que el algoritmo converja prematuramente en un mínimo local. Un ejemplo de mutación sería la alteración de un parámetro:

$$y_j = x_j + \epsilon$$

  donde ϵ es un pequeño valor aleatorio.

- **Selección de la próxima generación**: Después del cruce y la mutación, se selecciona la mejor descendencia para formar la siguiente generación, asegurando que las soluciones de mayor calidad se preserven o mejoren.

Los algoritmos genéticos son altamente robustos en funciones multimodales como la de Rosenbrock modificada, ya que su enfoque basado en la población y el uso de mutaciones les permite explorar el espacio de búsqueda globalmente y escapar de mínimos locales. Sin embargo, en casos de funciones no escalables como la de Rosenbrock modificada, el algoritmo puede ser menos eficiente debido a la naturaleza de la representación y el número de iteraciones necesarias para converger en poblaciones de mayor dimensión.

- **Ventajas:**
  - Muy robusto frente a mínimos locales.
  - Adecuado para problemas de gran dimensión o multimodales.
- **Desventajas:**
  - Puede requerir muchos parámetros y ser computacionalmente costoso.

Implementación:

```
import random
from deap import base, creator, tools, algorithms

def run_genetic_algorithm(x0=None):
    # Crear el tipo de problema de minimizacion
    creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMin)

    # Funcion para generar un individuo (solucion)
    def generate_individual():
        return [random.uniform(bounds[0][0], bounds[0][1]), random.uniform(bounds[1][0],
    bounds[1][1])]

    # Definir la funcion de evaluacion usando la funcion modificada de Rosenbrock
    def evaluate(individual):
        x1, x2 = individual
        return rosenbrock_modified([x1, x2]),

    # Configuracion de los operadores geneticos
    toolbox = base.Toolbox()
    toolbox.register("individual", tools.initIterate, creator.Individual,
    generate_individual)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("mate", tools.cxBlend, alpha=0.5)
    toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=0.2, indpb=0.2)
    toolbox.register("select", tools.selTournament, tournsize=3)
    toolbox.register("evaluate", evaluate)

    # Parametros del algoritmo genetico
    population_size = 100
    num_generations = 100
    mutation_prob = 0.2
    crossover_prob = 0.5

    # Crear la poblacion inicial
```

```
    population = toolbox.population(n=population_size)
    algorithms.eaSimple(population, toolbox, cxpb=crossover_prob, mutpb=mutation_prob, ngen=
    num_generations, verbose=False)

    # Obtener la mejor solucion
    best_individual = tools.selBest(population, k=1)[0]
    return best_individual
```

Resultado:
Solución hallada: $x^* = (-1{,}000227823671769, -1{,}000011113528561)$, $f(x^*) = -326{,}000227825111494$
Tiempo de ejecución: 0.208201 segundos

4.3. Algoritmos de Optimización basados en población

Los algoritmos basados en población como PSO mantienen una población de posibles soluciones que se mueven a través del espacio de búsqueda. Estos algoritmos son adecuados para funciones multimodales y pueden escapar de mínimos locales debido a la naturaleza colaborativa entre partículas.

4.3.1. Algoritmo de Enjambre de Partículas (*Particle Swarm Optimization* - PSO)

PSO es un algoritmo inspirado en el comportamiento de enjambres naturales, como bandadas de pájaros o cardúmenes de peces. Cada partícula en el enjambre representa una solución candidata al problema de optimización y se mueve a través del espacio de búsqueda basado en dos influencias principales: su mejor posición personal encontrada hasta ahora y la mejor posición global encontrada por el enjambre. El movimiento de cada partícula se actualiza según las siguientes reglas:

- **Actualización de la velocidad**:

$$v_i^{(t+1)} = w \cdot v_i^{(t)} + c_1 \cdot r_1 \cdot (p_i - x_i^{(t)}) + c_2 \cdot r_2 \cdot (g - x_i^{(t)})$$

  donde:

  - $v_i(t)$ es la velocidad de la partícula i en el paso t.
  - $x_i(t)$ es la posición actual de la partícula i.
  - p_i es la mejor posición personal encontrada por la partícula.
  - g es la mejor posición global encontrada por el enjambre.
  - w es el factor de inercia, que controla la influencia de la velocidad anterior.
  - c_1 y c_2 son los coeficientes de aprendizaje que controlan la influencia de las posiciones personal y global, respectivamente.
  - r_1 y r_2 son números aleatorios en el intervalo $[0, 1]$.

- **Actualización de la posición**:

$$x_i^{(t+1)} = x_i^{(t)} + v_i^{(t+1)}$$

Las partículas se mueven en el espacio de búsqueda siguiendo estas reglas, lo que les permite explorar diferentes áreas del espacio y explotar las mejores soluciones encontradas por el enjambre. PSO es particularmente efectivo en funciones multimodales, como la función de Rosenbrock modificada, ya que las partículas colaboran para escapar de los mínimos locales. Al no depender de derivadas, PSO es adecuado para funciones no diferenciables y no separables.

Sin embargo, la falta de escalabilidad de la función puede afectar el rendimiento de PSO, ya que este algoritmo tiende a ser más eficiente en problemas de mayor dimensión, donde puede aprovechar mejor la exploración en el espacio de soluciones. En una función de baja dimensionalidad, como la de Rosenbrock modificada, PSO puede converger rápidamente a soluciones subóptimas si el enjambre no mantiene suficiente diversidad.

- **Ventajas:**

- Rápido en convergencia.
- Robusto en problemas multimodales.

- **Desventajas:**
  - Puede quedar atrapado en mínimos locales sin la adecuada configuración de parámetros.

Implementación:

```
from pyswarm import pso

def run_pso(x0=None):  # No necesita punto inicial
    lb = [bounds[0][0], bounds[1][0]]
    ub = [bounds[0][1], bounds[1][1]]
    xopt, _ = pso(rosenbrock_modified, lb, ub)
    return xopt
```

Resultado:
Solución hallada: $x^* = (-1{,}000227998663365, -1{,}000010667523723)$, $f(x^*) = -326{,}000227824124636$
Tiempo de ejecución: 0.126058 segundos

4.3.2. *Covariance Matrix Adaptation Evolution Strategy* (CMA-ES)

CMA-ES es un algoritmo evolutivo avanzado que ajusta dinámicamente la forma y el tamaño de la distribución de búsqueda basándose en el comportamiento de la población de soluciones. A diferencia de PSO, que se enfoca en actualizar posiciones y velocidades de partículas, CMA-ES adapta una matriz de covarianza que modela las correlaciones entre las variables, lo que permite capturar mejor la geometría del espacio de búsqueda.

El algoritmo sigue los siguientes pasos:

- **Generación de nuevas soluciones**: Se generan λ soluciones aleatorias a partir de una distribución multivariante normal con una media μ y una matriz de covarianza C:

  $$x_k \sim \mathcal{N}(\mu, \sigma^2 C)$$

  donde σ es la amplitud de la búsqueda (tamaño de paso) y C es la matriz de covarianza adaptada.

- **Evaluación y selección**: Las soluciones generadas se evalúan con la función objetivo, y las mejores soluciones son seleccionadas para actualizar la distribución.

- **Actualización de la media y la matriz de covarianza**: La media μ de la población se actualiza hacia el centro de las mejores soluciones seleccionadas, mientras que la matriz de covarianza C se adapta según las variaciones observadas en las soluciones seleccionadas, capturando las direcciones de búsqueda más prometedoras.

  La fórmula de actualización de la matriz de covarianza es:

  $$C_{t+1} = (1 - c_\mu)C_t + c_\mu \sum_{i=1}^{\mu} w_i (y_i - \mu_t)(y_i - \mu_t)^T$$

  donde c_λ es una constante de aprendizaje y w_i son los pesos asignados a las mejores soluciones seleccionadas.

CMA-ES es extremadamente eficaz en funciones multimodales y no separables como la de Rosenbrock modificada, ya que adapta dinámicamente su estrategia de búsqueda según la estructura del espacio de soluciones. La matriz de covarianza permite al algoritmo capturar las correlaciones entre las variables de manera efectiva, lo que es crucial en funciones no separables donde las variables están interrelacionadas. Sin embargo, al tratarse de una función no escalable (específica de dos dimensiones), CMA-ES no puede aprovechar plenamente su capacidad para manejar problemas de alta dimensionalidad. Aunque es poderoso, su costo computacional es más elevado en comparación con PSO, debido a la necesidad de calcular y actualizar la matriz de covarianza en cada iteración.

- **Ventajas:**
  - Robusto frente a funciones complicadas, sin necesidad de derivadas.
- **Desventajas:**
  - Puede ser costoso en términos computacionales.

Implementación:

```
import cma

def run_cma_es(x0):
    # sigma0 es la desviacion inicial (amplitud de busqueda)
    sigma0 = 0.5

    # Crear la funcion adaptada a CMA-ES
    es = cma.CMAEvolutionStrategy(x0, sigma0, {'bounds': [[bounds[0][0], bounds[1][0]], [
    bounds[0][1], bounds[1][1]]]})

    # Extraer mejor resultado
    result = es.optimize(rosenbrock_modified).best.x
    return result
```

Resultado:
Punto inicial: $x_0 = (0, 0)$
Solución hallada: $x^* = (-1{,}000227822273706, -1{,}000011115278564)$, $f(x^*) = -326{,}000227825111494$
Tiempo de ejecución: 0.877510 segundos

4.4. Algoritmos de Búsqueda Estocástica

Los algoritmos de búsqueda estocástica como el recocido simulado y el salto de cuenca son efectivos para funciones multimodales como la de Rosenbrock modificada, debido a su capacidad para escapar de mínimos locales. Ambos algoritmos son independientes de gradientes, lo que los hace adecuados para funciones no diferenciables o no separables. Sin embargo, sus costos computacionales y dependencias de parámetros clave (como la tasa de enfriamiento o los saltos aleatorios) limitan su aplicabilidad en problemas de mayor dimensión o en funciones no escalables como la de Rosenbrock modificada en su forma actual.

4.4.1. Recocido Simulado (*Simulated annealing* (SA))

El recocido simulado es un algoritmo de optimización inspirado en el proceso físico de enfriamiento lento de metales. En el recocido físico, cuando un metal se enfría lentamente, sus átomos encuentran una estructura de energía mínima, lo que da lugar a un estado estable. El recocido simulado sigue un enfoque similar, explorando soluciones a través de un proceso que inicialmente acepta movimientos a soluciones peores, pero reduce esta probabilidad a medida que "enfría" el sistema.

El algoritmo funciona de la siguiente manera:

- **Perturbación de la solución**: En cada iteración, se genera una nueva solución x' a partir de la solución actual x, generalmente mediante una perturbación aleatoria:

$$x' = x + \epsilon$$

  donde ϵ es un valor aleatorio pequeño.

- **Aceptación de la solución**: Si la nueva solución es mejor, se acepta automáticamente. Si es peor, se acepta con una probabilidad P, dada por:

$$P = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

  donde $f(x)$ es la función objetivo, y T es la temperatura, que disminuye con el tiempo. Esta probabilidad decrece conforme la temperatura T disminuye, lo que permite al algoritmo escapar de mínimos locales en las etapas iniciales, pero refina la búsqueda en las etapas posteriores.

- **Enfriamiento**: La temperatura T se reduce siguiendo un calendario de enfriamiento, que puede ser lineal o exponencial. Por ejemplo:

$$T = T_0 \cdot \alpha^k$$

donde T_0 es la temperatura inicial, α es el factor de enfriamiento (por ejemplo, 0,95), y k es el número de iteraciones.

El recocido simulado es especialmente útil en funciones multimodales como la de Rosenbrock modificada, ya que su capacidad para aceptar soluciones subóptimas en las primeras etapas le permite escapar de mínimos locales. Al no depender de gradientes, puede lidiar con funciones no diferenciables y no separables. Sin embargo, como la función de Rosenbrock modificada no es escalable, el SA tiende a ser menos efectivo en problemas de alta dimensionalidad, donde se necesitarían muchas más iteraciones para encontrar una solución óptima. Además, el éxito del algoritmo depende críticamente de la tasa de enfriamiento y de la duración del proceso.

- **Ventajas:**
  - Capaz de escapar de mínimos locales.
  - Fácil de implementar.
- **Desventajas:**
  - Puede ser lento en converger hacia una solución óptima.
  - Requiere ajuste cuidadoso de parámetros.

Implementación:

```
from scipy.optimize import dual_annealing

def run_simulated_annealing(x0=None):  # No necesita punto inicial
    result = dual_annealing(rosenbrock_modified, bounds)
    return result.x
```

Resultado:
Solución hallada: $x^* = (-1{,}000227828504980, -1{,}000011120981859)$, $f(x^*) = -326{,}000227825111210$
Tiempo de ejecución: 0.287594 segundos

4.4.2. Método de Salto de Cuenca (*Basin Hopping*)

El método de salto de cuenca es un enfoque de optimización global diseñado para funciones multimodales. Combina una búsqueda local con saltos aleatorios a lo largo del espacio de soluciones para escapar de mínimos locales. La idea es realizar minimizaciones locales en cada iteración y, luego, realizar un "salto" a una nueva región del espacio de búsqueda, con la esperanza de encontrar un mínimo mejor en una nueva cuenca de atracción.

El algoritmo sigue los siguientes pasos:

- **Minimización local**: En cada iteración, se realiza una minimización local utilizando un algoritmo como BFGS o Nelder-Mead desde la posición actual x. Esto asegura que se explora la cuenca local de atracción alrededor de x y se llega al mínimo local x_{local}.

$$x_{\text{local}} = \min f(x)$$

- **Salto aleatorio**: Después de encontrar el mínimo local, el algoritmo realiza un salto aleatorio en el espacio de soluciones. El salto puede ser una perturbación aleatoria ϵ aplicada a x_{local}:

$$x_{\text{new}} = x_{\text{local}} + \epsilon$$

- **Aceptación de la nueva solución**: La nueva solución x_{new} se acepta si es mejor que la solución actual, o bajo un criterio estocástico similar al recocido simulado, permitiendo que el algoritmo escape de mínimos locales ocasionalmente.

El salto de cuenca es particularmente útil para funciones como la de Rosenbrock modificada, que presenta múltiples mínimos locales. A diferencia de otros métodos estocásticos, el salto de cuenca realiza una búsqueda local detallada antes de realizar saltos aleatorios, lo que lo convierte en una combinación de búsqueda local y global. Su enfoque basado en minimizaciones locales lo hace adecuado para funciones diferenciables como la de Rosenbrock modificada, donde los algoritmos locales como BFGS pueden aprovechar la suavidad de la función. Sin embargo, debido a que este método realiza múltiples minimizaciones locales, su costo computacional puede ser elevado, especialmente si la función no es escalable y el espacio de búsqueda no crece fácilmente con la dimensionalidad.

Al igual que el recocido simulado, el método de salto de cuenca es robusto frente a la multimodalidad y puede escapar de mínimos locales, pero su eficiencia depende de la frecuencia y magnitud de los saltos aleatorios. Su rendimiento también puede verse limitado si la función no se adapta bien a saltos en dimensiones más altas o si el problema no escalable no permite exploración significativa en dimensiones superiores.

- **Ventajas:**
  - Eficaz en funciones multimodales.
  - Combina búsqueda global y local.
- **Desventajas:**
  - El rendimiento depende de la calidad de la minimización local.
  - Puede ser computacionalmente costoso en problemas grandes.

Implementación:

```
from scipy.optimize import basinhopping

def run_basin_hopping(x0):
    # Definir minimizador local (BFGS)
    minimizer_kwargs = {"method": "L-BFGS-B", "bounds": bounds}
    result = basinhopping(rosenbrock_modified, x0, minimizer_kwargs=minimizer_kwargs, niter
    =100)
    return result.x
```

Resultado:
Punto inicial: $x_0 = (0, 0)$
Solución hallada: $x^* = (-1{,}000227825153447, -1{,}000011115491677)$, $f(x^*) = -326{,}000227825111438$
Tiempo de ejecución: 1.266273 segundos

5. Resultados y comparación

En la sección anterior se mostró para cada algoritmo el resultado de una sola ejecución de los mismos (con punto inicial $x_0 = (0, 0)$ en los que lo requerían). Para un análisis más exhaustivo se realizaron 20 ejecuciones de cada algoritmo con punto inicial distinto en cada una (aleatorio, dentro del espacio $-2 \leq x_1, x_2 \leq 2$) para los que lo requerían. Se guardaron todas las soluciones obtenidas, así como los tiempos de ejecución. A partir de estos se computan para cada algoritmo su mejor solución encontrada, su peor solución encontrada, su solución promedio, desviación estándar, y tiempo promedio de ejecución.

En la tabla 1 se resumen los resultados obtenidos (excepto la desviación estándar), y se marcan los mejores valores (verde), los peores valores (rojo), y los segundos mejores valores (subrayado).

Gracias a la implementación de los distintos algoritmos de optimización, se concluye que el mínimo global de la función dentro del espacio delimitado por $-2 \leq x_1, x_2 \leq 2$ se localiza aproximadamente en el punto $x^* = (x_1 = -1{,}000227822423090, x_2 = -1{,}000011114025018)$, donde la función toma el valor $f(x^*) = -326{,}000227825111494$. Este valor representa el mejor resultado obtenido en las ejecuciones. Aunque puntos aún más precisos en su vecindad podrían haberse encontrado incrementando el número de ejecuciones y las iteraciones de los algoritmos, se debe considerar que las limitaciones inherentes a la precisión de los valores de punto flotante y la estructura misma de la función hacen que las mejoras en la vecindad de

Algoritmo	Mejor Solución	Peor Solución	Solución Media	Tiempo Promedio (s)
Gradient Descent	72.997500000000002	74.978270696727506	74.330875074664007	0.052380084991455
BFGS	-326.000227825111438	73.433430666783252	-146.414583437930304	0.001103937625885
Trust Region	-326.000227825111494	72.998304032189722	-106.551388864023735	0.265096449851990
Nelder-Mead	-326.000227825111438	72.997500000000130	-226.250956790353030	0.005997836589813
Differential Evolution	-326.000227825111324	-326.000227820604323	-326.000227824779927	0.041150748729706
Genetic Algorithm	-326.000227825111494	-326.000224186399009	-326.000227643175833	0.203493130207062
PSO	-326.000227824922149	-326.000227669227286	-326.000227799783829	0.117109680175781
CMA-ES	-326.000227825111494	72.997500000000002	-246.200682260089110	1.150205373764038
Simulated Annealing	-326.000227825111438	-326.000227824235196	-326.000227825027309	0.215206158161163
Basin Hopping	-326.000227825111438	-326.000227825111324	-326.000227825111438	0.280462110042572

Cuadro 1: Resultados de los algoritmos de optimización. Marcados en los mejores valores (verde), los peores valores (rojo), y los segundos mejores valores (subrayado).

este punto sean insignificantes. Estas limitaciones van más allá del alcance de este estudio y no alteran sustancialmente las conclusiones del mismo.

Se resumen además los resultados en 5 gráficas de barras (ver gráfica 3) representando para cada algoritmo entre todas las ejecuciones: el menor error absoluto con respecto al mínimo global conocido, el error absoluto promedio con respecto al mínimo global conocido, el mayor error absoluto con respecto al mínimo global conocido, la desviación estándar del error, y el tiempo promedio de ejecución. Y adicionalmente se grafican en conjunto de forma comparativa la Mejor Precisión vs Tiempo Promedio de Ejecución (gráfica 4) y a Precisión Promedio vs Tiempo Promedio de Ejecución (gráfica 5).

5.1. Análisis individuales

- **Gradient Descent**: El método de descenso por gradiente muestra una capacidad limitada para encontrar el mínimo global en la función modificada de Rosenbrock. Su tendencia a quedar atrapado en mínimos locales se refleja en la diferencia notable entre las soluciones óptimas y las subóptimas, con una eficiencia de tiempo relativamente baja. Esto es consistente con la naturaleza de la función, que es continua y diferenciable pero presenta múltiples mínimos locales, lo que hace que el gradiente sea menos efectivo.

- **BFGS**: El método BFGS, una técnica de optimización basada en gradientes, tiene un desempeño superior en términos de hallar soluciones cercanas al mínimo global. Su capacidad para encontrar buenas soluciones con tiempos de ejecución significativamente menores sugiere que es efectivo en explorar el espacio de búsqueda de manera más eficiente. Sin embargo, su tendencia a quedarse atrapado en mínimos locales es evidente en la diferencia entre la mejor y la peor solución encontrada.

- **Trust Region**: El método de región de confianza muestra una capacidad robusta para encontrar soluciones cercanas al mínimo global. Aunque su tiempo de ejecución es mayor en comparación con otros métodos, su rendimiento en términos de soluciones es sólido, con una mejor capacidad para escapar de los mínimos locales. Este algoritmo maneja bien la característica multimodal de la función, ajustando su exploración en regiones específicas del espacio de búsqueda.

- **Nelder-Mead**: Nelder-Mead ofrece una buena capacidad para encontrar soluciones cercanas al mínimo global, con tiempos de ejecución relativamente rápidos. Su método de búsqueda basado en la simplex es menos dependiente de los gradientes, lo que le permite manejar la naturaleza no lineal y multimodal de la función modificada de Rosenbrock con relativa eficacia. Sin embargo, su desempeño puede verse afectado por la no separabilidad de la función.

- **Differential Evolution**: La evolución diferencial se destaca por su capacidad para mantener la calidad de las soluciones a lo largo de múltiples ejecuciones, encontrando consistentemente soluciones cercanas al mínimo global. Su robustez en la exploración del espacio de búsqueda, gracias a su mecanismo de mutación y cruzamiento, le permite escapar de mínimos locales y manejar la naturaleza multimodal de la función con buena eficiencia de tiempo.

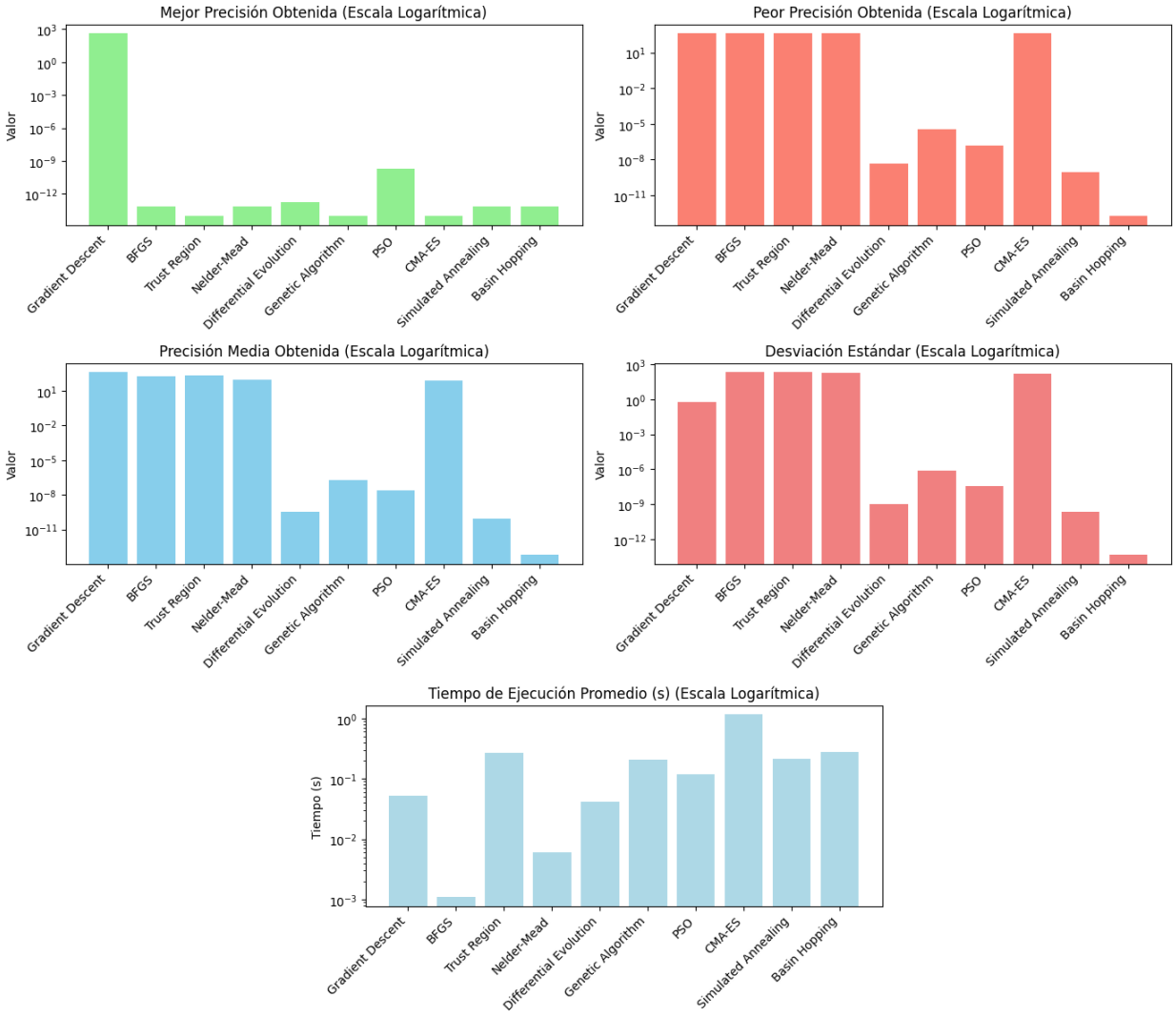

Figura 3: Resumen de resultados.

- **Genetic Algorithm**: El algoritmo genético también muestra un rendimiento robusto al encontrar soluciones cercanas al mínimo global. Su capacidad para explorar el espacio de búsqueda utilizando operaciones de selección, cruzamiento y mutación le permite superar los mínimos locales de manera efectiva. Sin embargo, su tiempo de ejecución es mayor en comparación con algunos otros algoritmos, lo que refleja el costo de la exploración exhaustiva.
- **PSO**: La optimización por enjambre de partículas (PSO) ofrece un buen equilibrio entre calidad de solución y eficiencia de tiempo. Su enfoque basado en la colaboración de múltiples soluciones potenciales le permite explorar eficazmente el espacio de búsqueda, logrando resultados cercanos al mínimo global. Su capacidad para manejar la naturaleza multimodal y no escalable de la función es evidente en sus resultados consistentes.
- **CMA-ES**: La estrategia de adaptación de la matriz de covarianza (CMA-ES) tiene un rendimiento mixto, mostrando una buena capacidad para encontrar soluciones cercanas al mínimo global, pero con tiempos de ejecución significativamente mayores. Su enfoque basado en la adaptación de la covarianza permite una exploración exhaustiva del espacio de búsqueda, pero su alta complejidad computacional puede limitar su aplicabilidad en escenarios con restricciones de tiempo.

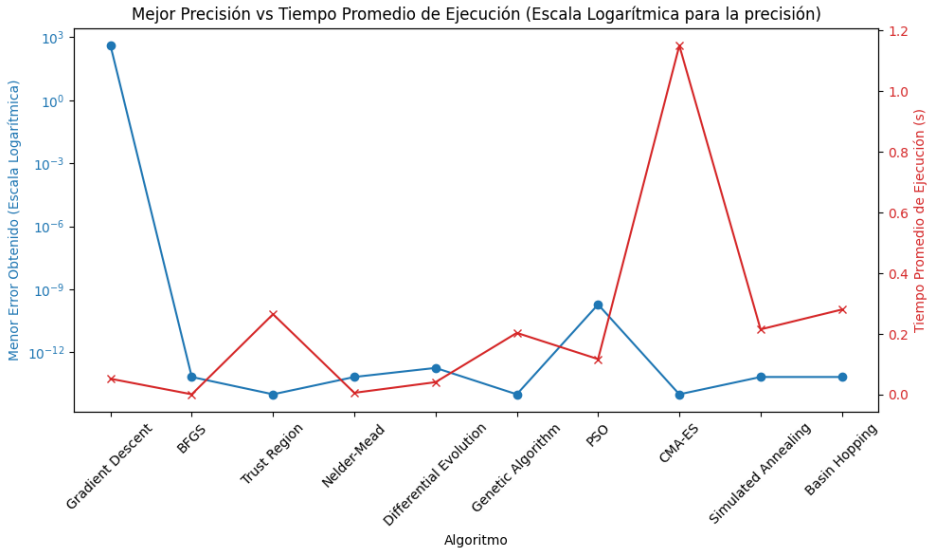

Figura 4: Gráfica de Mejor Precisión vs Tiempo Promedio de Ejecución.

- **Simulated Annealing**: El recocido simulado muestra una capacidad sólida para encontrar soluciones cercanas al mínimo global. Su enfoque estocástico le permite escapar de mínimos locales, manejando bien la naturaleza multimodal de la función. Aunque sus tiempos de ejecución son moderados, su capacidad para explorar el espacio de búsqueda de manera flexible es una ventaja.
- **Basin Hopping**: El método de salto de cuenca combina características de búsqueda local con una exploración global, lo que le permite encontrar soluciones cercanas al mínimo global de manera efectiva. A pesar de su mayor tiempo de ejecución, su capacidad para combinar exploración y explotación hace que sea un buen candidato para problemas con múltiples mínimos locales.

5.2. Resumen de Mejores Resultados

- **Mejores Resultados en la Solución**:
  Los algoritmos como BFGS, Trust Region, Nelder-Mead, Differential Evolution, Genetic Algorithm, PSO, Simulated Annealing, y Basin Hopping tienen un rendimiento sobresaliente en la obtención de soluciones cercanas al mínimo global. Los algoritmos evolutivos (Differential Evolution y Genetic Algorithm) y los métodos de búsqueda estocástica (Simulated Annealing y Basin Hopping) se destacan en la calidad de las soluciones.
- **Mejores Resultados en Tiempo de Ejecución**:
  BFGS es notable por su alta eficiencia, encontrando soluciones de calidad en el menor tiempo. Nelder-Mead y Simulated Annealing también presentan buenos tiempos de ejecución, destacándose por equilibrar la calidad de la solución con la eficiencia temporal.

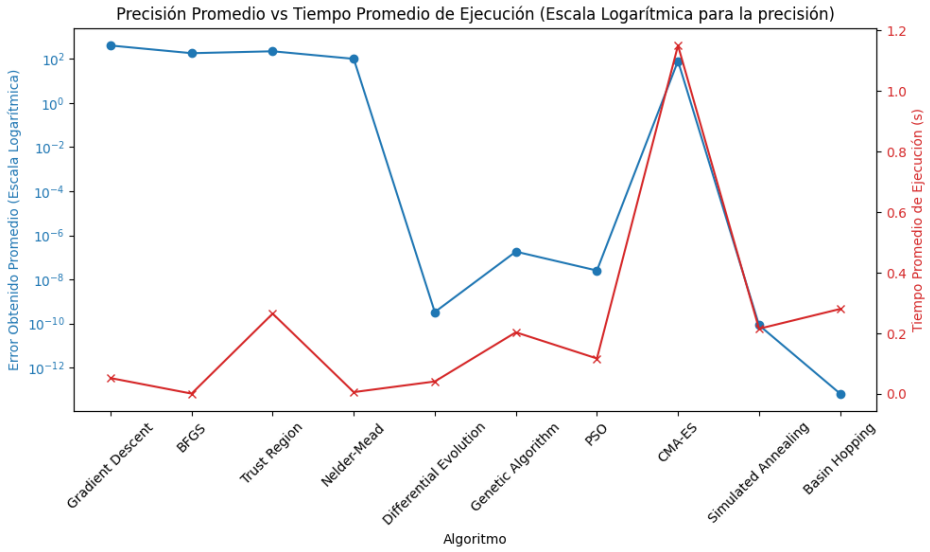

Figura 5: Gráfica de Precisión Promedio vs Tiempo Promedio de Ejecución.

6. Conclusiones

Mediante el análisis exhaustivo de diferentes algoritmos de optimización aplicados a la función de Rosenbrock modificada, se ha demostrado que la capacidad de cada algoritmo para encontrar el mínimo global y su eficiencia computacional varía significativamente, reflejando las características complejas de la función, que es continua, diferenciable, no separable, no escalable y multimodal.

Los algoritmos evolutivos como la Evolución Diferencial y el Algoritmo Genético, así como los métodos de búsqueda estocástica como el Recocido Simulado y el Salto de Cuenca, han mostrado una notable capacidad para encontrar soluciones cercanas al mínimo global. Estos métodos son robustos en explorar el espacio de búsqueda y evitar mínimos locales debido a sus mecanismos de mutación, cruzamiento y aceptación de soluciones subóptimas, lo que les permite manejar eficazmente la naturaleza multimodal y no escalable de la función de Rosenbrock.

En términos de eficiencia, el método BFGS se ha destacado por su rapidez en encontrar soluciones de alta calidad. A pesar de ser un método basado en gradientes, que enfrenta dificultades al tratar con múltiples mínimos locales, su capacidad para converger rápidamente lo convierte en una opción efectiva para problemas de optimización en entornos menos complejos. Otros métodos como Nelder-Mead y el Recocido Simulado también ofrecen un buen equilibrio entre calidad de solución y tiempo de ejecución, haciendo que sean opciones viables en escenarios prácticos.

Métodos como CMA-ES, aunque efectivos en términos de encontrar soluciones cercanas al mínimo global, enfrentan desafíos significativos en términos de tiempo de ejecución. La alta complejidad computacional asociada con la adaptación de la covarianza puede limitar su aplicabilidad en situaciones donde los recursos de tiempo son críticos. Por otro lado, algoritmos como el Descenso por Gradiente, a pesar de ser menos eficaces en encontrar el mínimo global, ofrecen una ejecución rápida, lo que puede ser útil en aplicaciones donde la calidad de la solución es menos crítica que el tiempo de respuesta.

La elección del algoritmo de optimización depende de las necesidades específicas del problema en cuestión. Los métodos evolutivos y de búsqueda estocástica son recomendables para problemas complejos con múltiples mínimos locales y alta dimensionalidad, mientras que los métodos basados en gradientes y de región de confianza pueden ser adecuados para problemas con menos complejidades. El análisis sugiere que una combinación de métodos podría ser la mejor estrategia, utilizando métodos rápidos para exploraciones preliminares y métodos más robustos para refinamientos finales.

En resumen, el estudio ha proporcionado una visión clara de cómo diversos algoritmos de optimización abordan el problema de la función de Rosenbrock modificada, destacando la importancia de seleccionar el método adecuado basado en el equilibrio entre la calidad de la solución y la eficiencia computacional.