MODELO DE OPTIMIZACIÓN CON ENJAMBRE DE PARTÍCULAS

Gustavo Amador Fonseca - C20451

Universidad de Costa Rica

25 de Junio del 2024

Contenidos

Introducción

Para resolver problemas de optimización se pueden usar tecnicas exáctas y heurísticas.

«Técnica basada en el conocimiento del problema que se está tratando, realiza una búsqueda inteligente en el espacio, que le permite encontrar un grado alto de confianza en la solución» (Erroz, 2022)

El PSO es un método de optimización heurística orientado a encontrar mínimos o máximos globales.

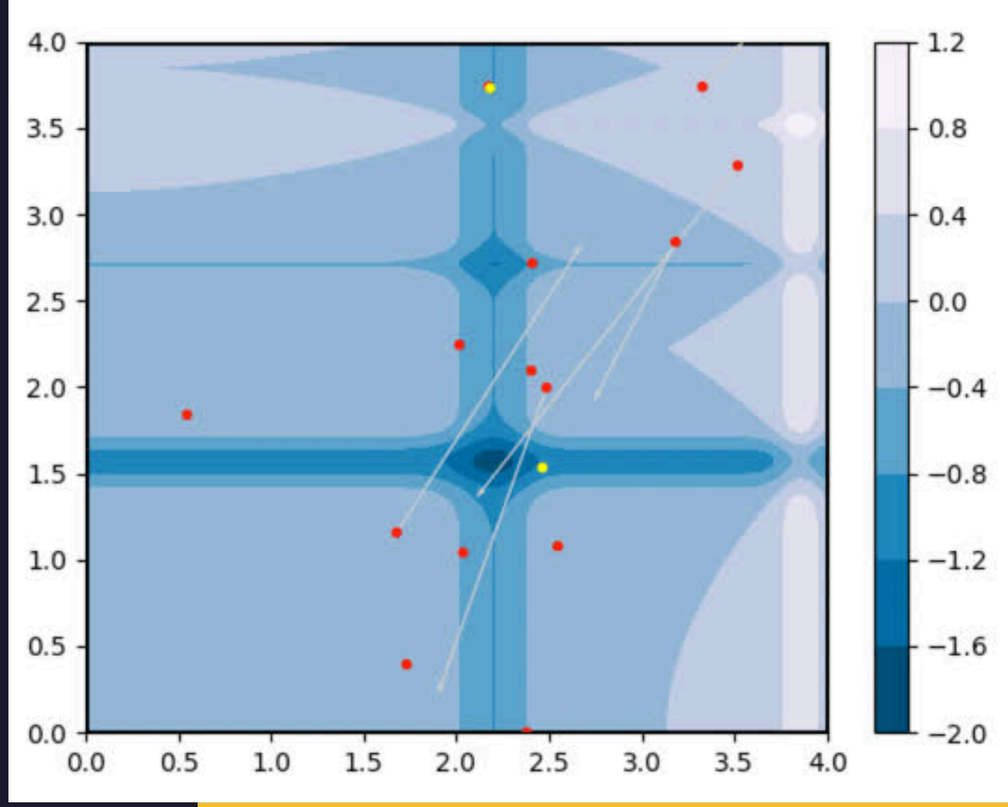
Desarrollado por James Kennedy y Russell Eberhart en 1995 mientras estudiaban un modelo para describir el comportamiento social de los animales en grupo.

Es inspirado en el comportamiento de los ejambres en la naturaliza, depende del azhar y el comportamiento ante la comunicación entre partículas del ejambre.

OBJETIVOS

Objetivo General

Programar y mejorar el modelo de optimización con ejmabre de partículas mediante paralelización en el lenguaje de programación Python.



Objetivos Específicos

Analizar y programar el algoritmo PSO estándar planteado en el trabajo de maestría de David Erroz Arroyo.

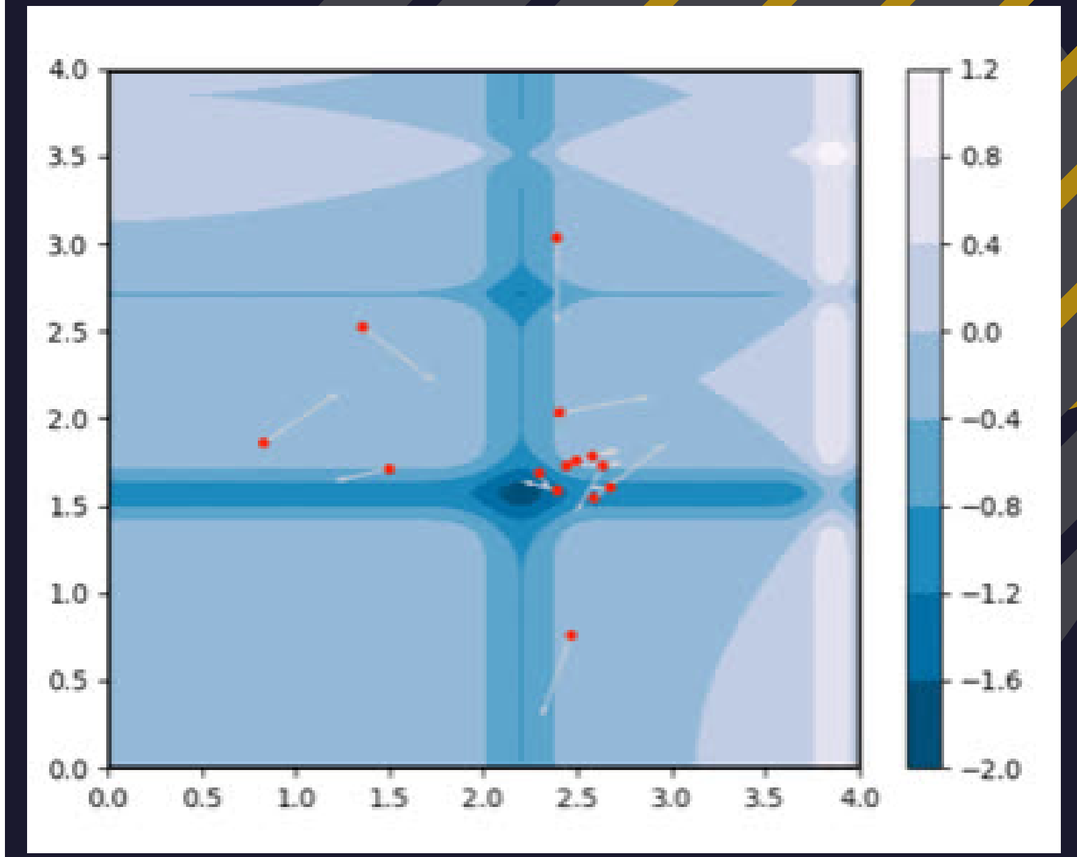
Desarrollar un modelo PSO mediante técnicas de paralelización en Python para mejorar la convergencia en el espacio.

Comparar cuantitativa los modelos PSO estándar, paralelizado y un método de optimización basado en gradiente, evaluando su desempeño en términos de velocidad y precisión.

MODELO DE OPTIMIZACIÓN POR EJAMPRE DE PARTÍCULAS ESTÁNDAR

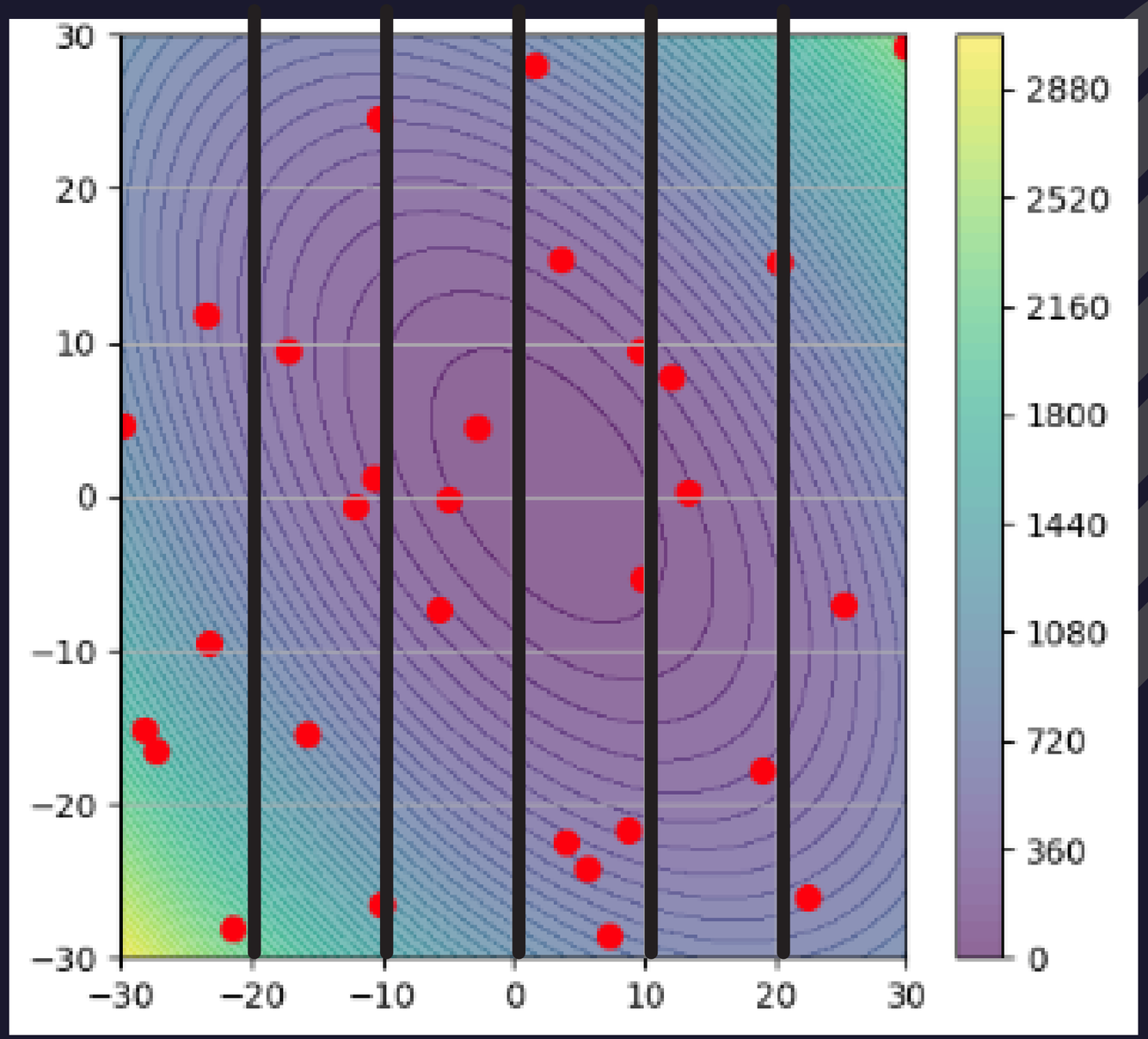
Algoritmo 1: Algoritmo PSO estándar

para *cada partícula* $i = 1, ..., N$ **hacer**
  Inicializar la posición X_i de la partícula dentro de los límites: $x_{i,j} \in [l_j, u_j]$
  Inicializar la mejor posición individual $pBest_i$ a la posición inicial: $pBest_i = X_i$
  si $f(pBest_i) < f(gBest)$ **entonces**
    Actualizar la mejor posición colectiva: $gBest = pBest_i$
  fin
  Inicializar la velocidad V_i de la partícula
fin
mientras *No se cumpla el criterio de terminación* **hacer**
  para *cada partícula* $i = 1, ..., N$ **hacer**
    Actualizar la velocidad V_i de la partícula según la ecuación (2.2) $\forall\, j = 1, ..., d$
    Actualizar la posición X_i de la partícula según la ecuación (2.3) $\forall\, j = 1, ..., d$
    Comprobar y gestionar si es necesario la factibilidad de las posiciones X_i
  fin
  para *cada partícula* $i = 1, ..., N$ **hacer**
    si $f(X_i) < f(pBest_i)$ **entonces**
      Actualizar la mejor posición individual: $pBest_i = X_i$
      si $f(pBest_i) < f(gBest)$ **entonces**
        Actualizar la mejor posición colectiva: $gBest = pBest_i$
      fin
    fin
  fin
fin



$$v_{i,j} = w * v_{i,j} + c_1 * rand_1 * (pbest_{i,j} - x_{i,j}) + c_2 * rand_2 * (gbest_j - x_{i,j}) \qquad (2.2)$$

MODELO DE OPTIMIZACIÓN POR EJAMPRE DE PARTÍCULAS PARALELIZADO

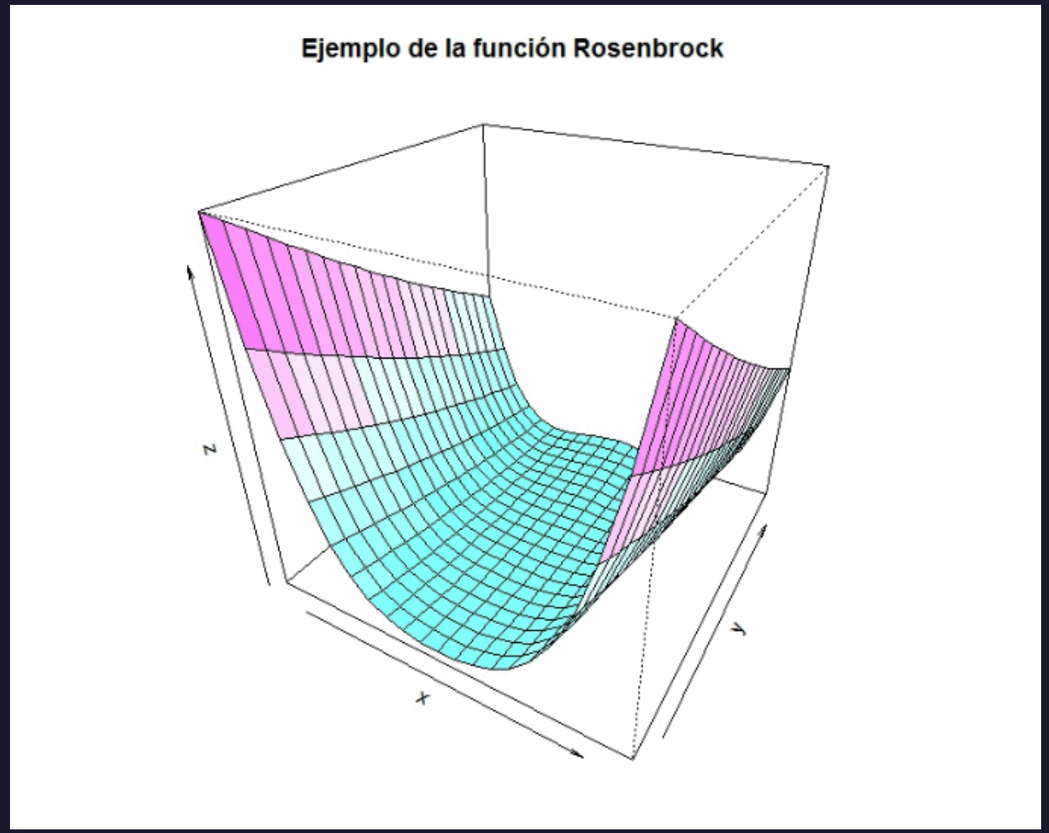

30
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20
30
2880
2520
2160
1800
1440
1080
720
360
0

```python
        return resultado

    def ejecutar_pso_en_subregion(self, lim_inf, lim_sup, funcion, n_particulas, max_iter):
        if funcion == 'ackley':
            return self.pso(self.funcion_ackley, lim_inf, lim_sup, n_particulas=n_particulas, max_iter=max_iter)
        elif funcion == 'cuadratica':
            if self.dimensiones != 2:
                raise ValueError("La cuadrática solo acepta 2 dimensiones.")
            return self.pso(self.funcion_cuadratica, lim_inf, lim_sup, n_particulas=n_particulas, max_iter=max_iter)
        else:
            raise ValueError("Función ingresada inválida.")

    def dividir_espacio_de_busqueda(self):
        subregiones = []
        #https://www.datacamp.com/tutorial/how-to-use-the-numpy-linspace-function?utm_source=google&utm_medium=paid_search&utm
        #_campaignid=21057859163&utm_adgroupid=157296744657&utm_device=c&utm_keyword=&utm_matchtype=&utm_network=g&utm_adpostion
        #=&utm_creative=703052950521&utm_targetid=dsa-2218886984100&utm_loc_interest_ms=&utm_loc_physical_ms=1003683&utm_content=&utm
        #_campaign=230119_1-sea~dsa~tofu_2-b2c_3-es-lang-en_4-prc_5-na_6-na_7-le_8-pdsh-go_9-nb-e_10-na_11-na-june24&gad_source=1&gcl
        #id=CjwKCAjw1emzBhB8EiwAHwZZxRwySy9mn84bP4Lb3EBjWEqiU9rEej0jwbMd-LvqxNCcyxOFsHh0aBoCOC8QAvD_BwE
        intervalos = np.linspace(self.lim_inf_global, self.lim_sup_global, self.num_subregiones + 1)
        for i in range(self.num_subregiones):
            subregiones.append((intervalos[i], intervalos[i + 1]))
        return subregiones

    def ejecutar_optimizacion(self, n_particulas=30, max_iter=100):
        inicio = time.time()

        # Crear los límites de cada subregión
        subregiones = self.dividir_espacio_de_busqueda()

        # Ejecutar PSO en paralelo en cada subregión
        #https://docs.python.org/3/library/concurrent.futures.html
        with concurrent.futures.ProcessPoolExecutor() as executor:
            futures = [executor.submit(self.ejecutar_pso_en_subregion, lim_inf, lim_sup, self.funcion, n_particulas, max_iter) for lim_inf, lim_sup in sub
            resultados = [future.result() for future in concurrent.futures.as_completed(futures)]

        # Encontrar la mejor solución entre todas las subregiones
        mejor_resultado = min(resultados, key=lambda x: x[1])
        fin = time.time()

        mejor_valor = mejor_resultado[1]
        tiempo = fin - inicio
        return mejor_valor, tiempo
        #print(f'Número de iteración: {mejor_resultado[2]}')
        #print(f"Mejor posición: {mejor_resultado[0]}")
        #print(f"Mejor valor: {mejor_resultado[1]}")

        #print(f'Tiempo transcurrido: {fin - inicio} segundos')

# Ejemplo de uso
if __name__ == '__main__':
    paralelizacion = Paralelizacion(dimensiones=5, lim_inf_global=-30.0, lim_sup_global=30.0, num_subregiones=6, funcion='ackley')
    paralelizacion.ejecutar_optimizacion(n_particulas=50, max_iter=100)
```

MODELO DE OPTIMIZACIÓN POR EJAMPRE DE PARTÍCULAS PARALELIZADO

Broyden-Fletcher-Goldfarb-Shanno

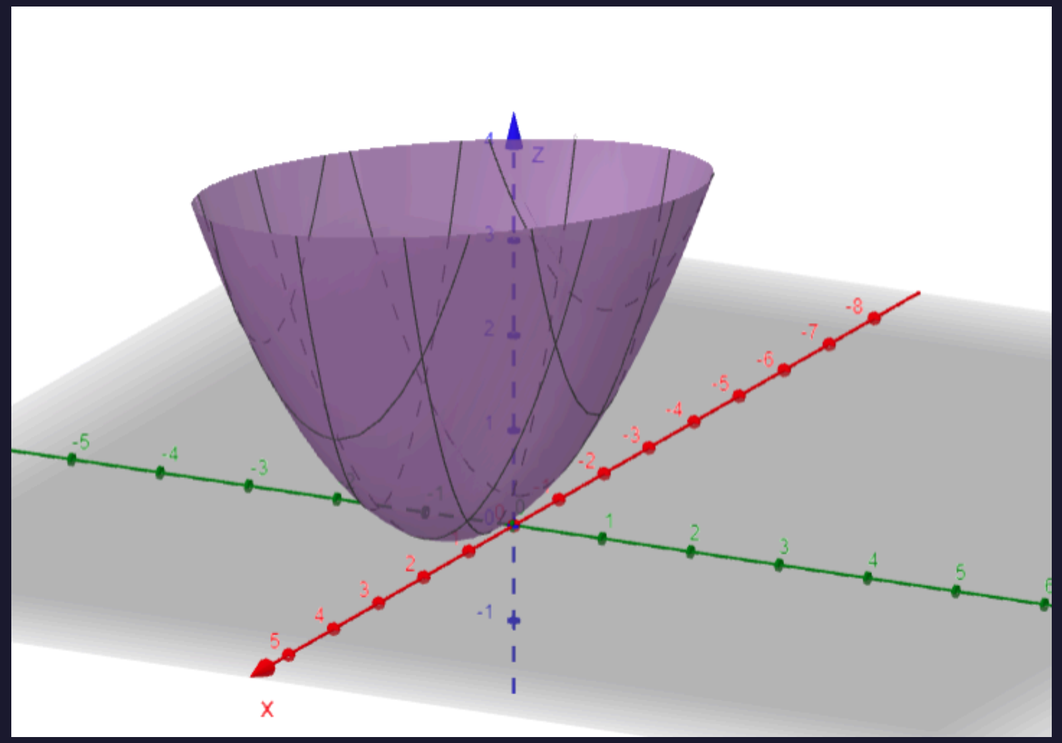
- BFGS es un algoritmo de optimización que busca el mínimo de una función utilizando tanto el gradiente como una aproximación de la Hessiana.
- Itera actualizando el punto actual, la dirección de búsqueda y la aproximación de la Hessiana inversa.
- Utiliza búsquedas en línea para determinar el tamaño de paso óptimo en cada iteración.
- Converge cuando se cumple un criterio de parada, como un cambio pequeño en la función objetivo o el gradiente.



COMPARACIÓN DE MODELOS

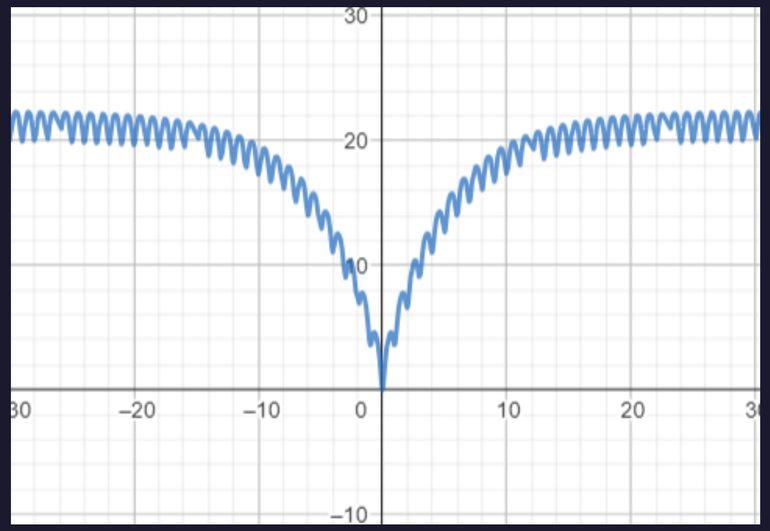
Función cuadrática

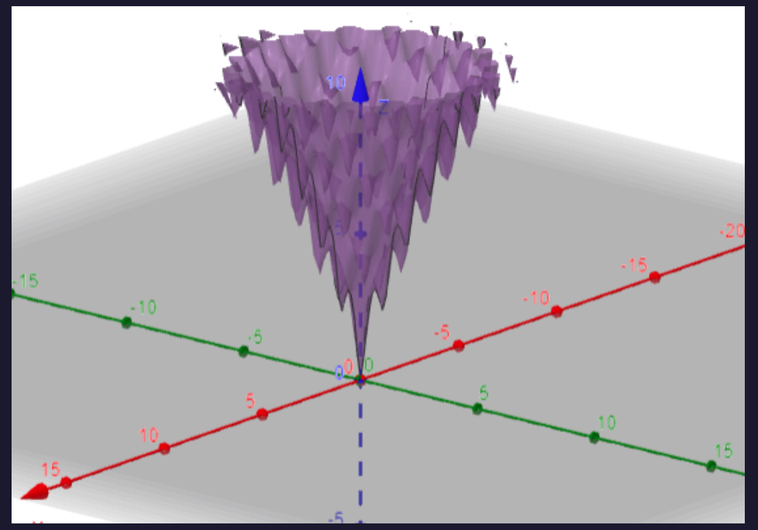
```python
def funcion_cuadratica(x):

    # Coeficientes cuadráticos
    A = np.array([[2, 1], [1, 2]])  # Matriz de coeficientes cu
    # Coeficientes lineales
    C = np.array([-6, -4])  # Vector de coeficientes lineales
    # Término constante
    D = 10  # Término constante

    # Calcular el valor de la función cuadrática
    resultado = 0.5 * np.dot(x.T, np.dot(A, x)) + np.dot(C, x)
    return resultado
```

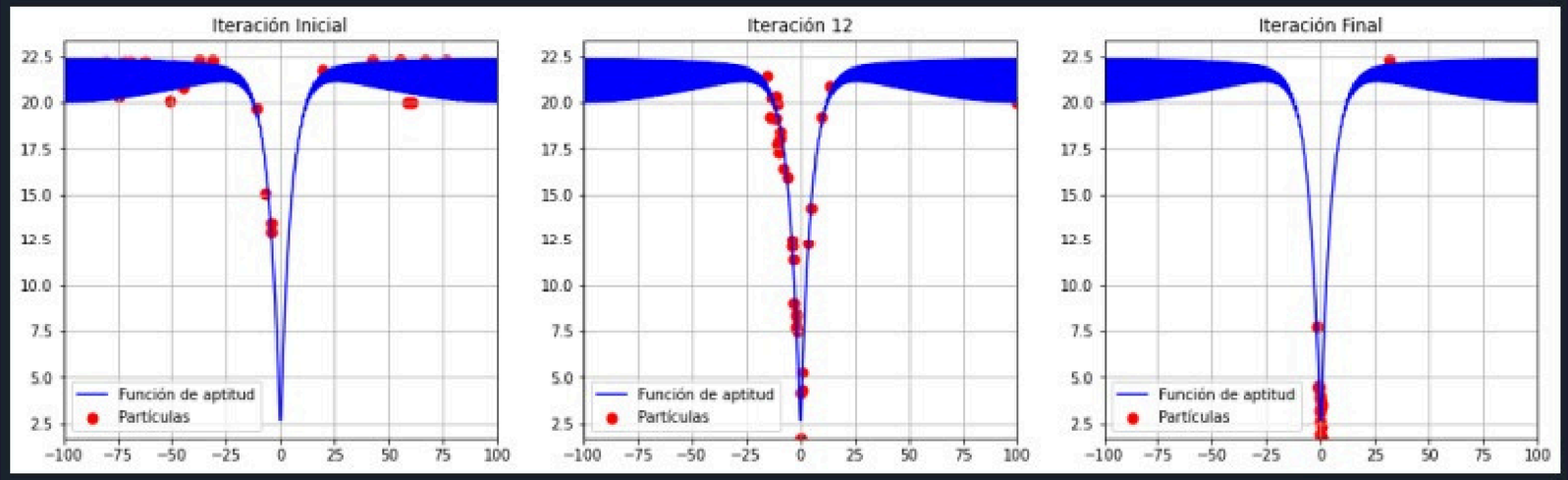


Función Ackley

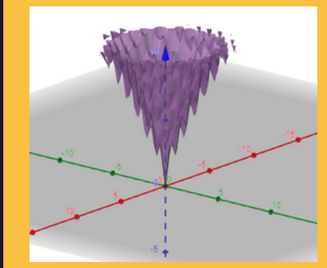
$$f(x,y) = -a \cdot \exp\left(-b \cdot \sqrt{\tfrac{1}{2}(x^2+y^2)}\right) - \exp\left(\tfrac{1}{2}(\cos(c \cdot x) + \cos(c \cdot y))\right) + a + \exp(1)$$



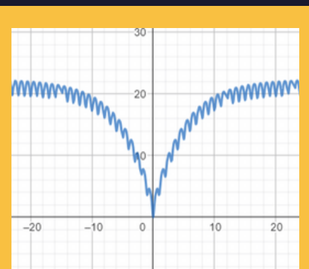




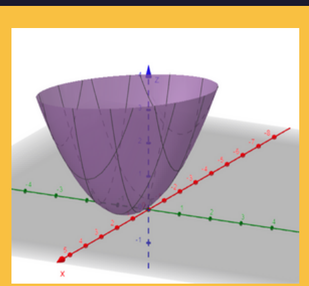
Conclusiones



Modelo PSO paralelizado superó ampliamente al PSO estándar y al modelo BFGS en la función Ackley.



Un umento en el número de iteraciones y partículas no implica una mejora directa en el rendimiento del modelo PSO paralelizado.



El aumento del intervalo de búsqueda baja considerablemente la efectividad del modelo.

Recomendaciones

Mejora

este modelo se puede hacer mucho más robusto agregando un segundo nivel de búsqueda más acotada al rededor de los puntos locales encontrados y asé aumentar la probabilidades de éxito.

Uso de librerías en GitHub

Durante mi investigación encontré muchas repositoriso con el modelo PSO y no solo eso, sino que librerías completas de metodos de optimización por partículas.

PySwarms

Es una librería ampliamente aceptada por la comunidad de python, que permite ejecutar el modelo PSO.

¿PREGUNTAS?