Tarea 3
Optimización de enjambre de partículas

Mario Emilio Jiménez Vizcaíno
A01173359@itesm.mx

1 INTRODUCCIÓN

El algoritmo de optimización de enjambre de partículas es un algoritmo basado en las técnicas de cómputo evolutivo, primera vez descrito en 1995 por Kennedy y Eberhart [1]. La intención original era simular gráficamente el vuelo de una parvada de pájaros, y durante la evolución del algoritmo surgió el objetivo de utilizarlo como un algoritmo optimizador para funciones continuas no lineales.

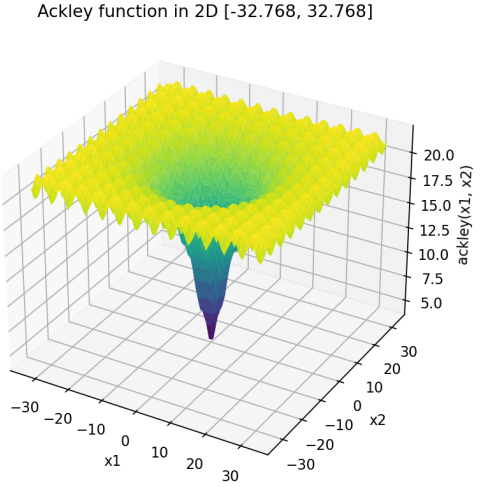
Durante la ejecución de este algoritmo un número de entidades, las partículas, se colocan en el espacio de búsqueda de la función y se calcula la evaluación del función en la posición actual de cada punto. Cada partícula entonces determina su movimiento a través del espacio de búsqueda combinando algún aspecto de la historia de la propia partícula y de la partícula con mejor evaluación. Eventualmente se espera que todas las partículas se acercen a un punto óptimo de la función. [2]

2 METODOLOGÍA

Para la demostración de este algoritmo se utilizó la función Ackley, cuya fórmula general es:

$$f(x) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Esta función es usada frecuentemente para probar algoritmos de optimización ya que alrededor del punto mínimo global (la coordenada $(0, 0)$) la gráfica presenta muchos puntos mínimos locales, por lo que los algoritmos de optimización corren el riesgo de quedarse atrapados en uno de estos. En la siguiente imagen se presenta el comportamiento de esta función en dos dimensiones:



Esta función fue evaluada (y graficada) con coordenadas de dos dimensiones, en el rango $-32.768 < x_1, x_2 < 32.768$ con los parámetros $a = 20$, $b = 0.2$ y $c = 2\pi$.

La implementación de la fórmula para listas *np.ndarray* de forma (n, d) puede ser encontrada en el apéndice A.

2.1 Representación de los individuos

Las partículas fueron representadas por listas *np.ndarray* de dos elementos, y concatenadas verticalmente, para formar una sola lista de forma $(n, 2)$, en donde n es el número de partículas en el enjambre (por defecto 500).

2.2 Algoritmo de optimización de enjambre de partículas

El algoritmo descrito en esta tarea fue implementado en dentro de una clase `ParticleSwarmOptimization`, con dos funciones principales: el constructor y la función `run`, descritas a continuación.

La implementación del algoritmo puede ser encontrada el apéndice B.

2.2.1 *Constructor de la clase* `ParticleSwarmOptimization`.

La primera función sólo sirve para inicializar los argumentos del algoritmo, como son la función de evaluación, el rango de evaluación, el número de partículas y de iteraciones, α, β, la rapidez máxima y una semilla para el generador de números aleatorios.

2.2.2 *Función* `run` *de la clase* `ParticleSwarmOptimization`.

En esta función se encuentra el algoritmo en sí, y comienza generando las posiciones de las partículas al azar, guardando una copia de estas como mejores locales, e inicializando la velocidad de las partículas en 0.

Después, por cada iteración del algoritmo:

(1) Evalúa las partículas usando la función proveída en el constructor (en nuestro caso la función Ackley)
(2) Actualiza la partícula con mejor evaluación global
(3) Actualiza las partículas locales mejores, comparando la evaluación de las partículas locales mejores con la evaluación de las partículas actuales
(4) Actualiza la velocidad de las partículas utilizando α, β, dos coordenadas aleatorias (ϵ_1 y ϵ_2), y los mejores globales (g^*) y locales (x^*) en la siguiente fórmula

$$v_{t+1} = v_t + \alpha\epsilon_1(g^* - x_t) + \beta\epsilon_2(x^* - x_t)$$

(5) Reduce las velocidades cuya magnitud superan la rapidez máxima
(6) Actualiza las posiciones de las partículas

$$x_{t+1} = x_t + v_{t+1}$$

Finalmente, la función regresa el historial de la posición de la partícula que tuvo mejor evaluación en cada iteración para poder visualizar el comportamiento del algoritmo.

3 RESULTADOS

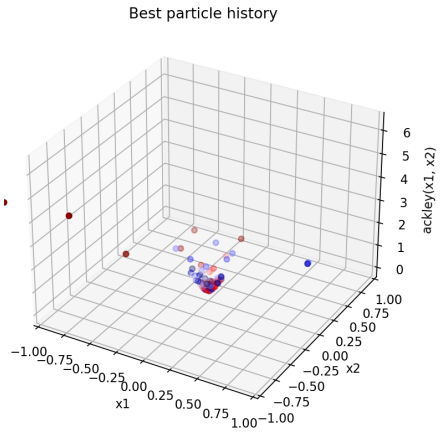
Para esta demostración se usaron los parámetros por defecto de la clase:

- número de partículas = 500
- número de iteraciones = 100
- α = 2
- β = 2
- v_{max} (rapidez máxima) = 2

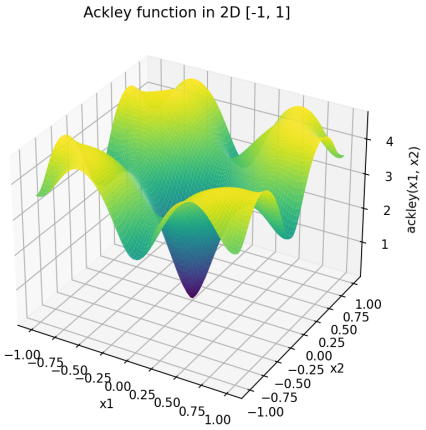
El único parámetro que se especifica además de la función de evaluación y el rango es `seed`, inicializado como 0 para que el experimento sea reproducible.

El tiempo total de ejecución es de 2.17 segundos, con la mejor partícula de todas las iteraciones en la posición (0.008138, -0.001466), con una evaluación de 0.025207, encontrada en la iteración 67.

La siguiente gráfica muestra la posición de la mejor partícula en cada evaluación, representando las primeras iteraciones con puntos azules y las últimas con puntos rojos.

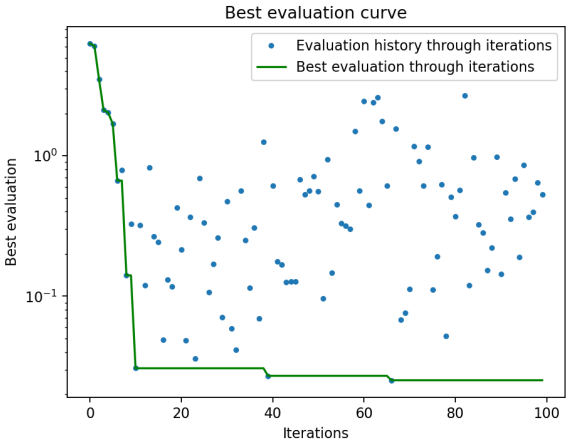


Se puede observar la similaridad de esta gráfica con la gráfica de la función Ackley en el mismo rango:



3.1 Curva de mejor encontrado

También se incluye una gráfica de la mejor evaluación encontrada por cada iteración. El eje y se graficó en una escala logarítmica para apreciar la mejora muy acelerada en las primeras iteraciones, y después de la iteración 10 mejoras ocasionalmente.



Las mejores evaluaciones se presentan como puntos azules, y además la línea verde muestra el mejor encontrado a través de todas las iteraciones.

4 CONCLUSIONES Y RETOS ENCONTRADOS

En esta práctica se demostró la utilidad y la implementación de uno de los algoritmos cuya creación fue relativamente reciente, además de cómo se comporta cuando las partículas se acercan cada vez más al punto óptimo de la función. Considero que este tipo de algoritmos tienen un muy buen uso para optimización de funciones pero me hubiera gustado profundizar más en las diferentes formas de determinar cuándo parar el algoritmo en vez de sólo usar un número predefinido de iteraciones.

El reto principal de la implementación pienso que fue diseñar las gráficas en tres dimensiones y crear una implementación con buen desempeño para seleccionar y actualizar los mejores puntos locales en cada iteración.

REFERENCES

[1] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, Vol. 4. IEEE, 1942–1948.

[2] Riccardo Poli, James Kennedy, and Tim Blackwell. 2007. Particle swarm optimization. *Swarm intelligence* 1, 1 (2007), 33–57.

A IMPLEMENTACIÓN DE LA FUNCIÓN ACKLEY PARA VECTORES *NP.NDARRAY*

```python
import numpy as np


def ackley_fun(v: np.ndarray) -> np.ndarray:
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = v.shape[1]
    term1 = -a * np.exp(-b * np.sqrt(np.square(v).sum(1) / d))
    term2 = -np.exp(np.cos(c * v).sum(1) / d)
    return term1 + term2 + a + np.e


if __name__ == "__main__":
    import matplotlib.pyplot as plt

    def generate_graph(lim: float):
        divs = 80
        x = np.linspace(-lim, lim, divs)
        y = np.linspace(-lim, lim, divs)
        X, Y = np.meshgrid(x, y)
        Z = ackley_fun(np.dstack((X, Y)).reshape(-1, 2))
        Z = Z.reshape(X.shape[0], -1)

        plt.figure(figsize=(6, 5), dpi=160)
        ax = plt.axes(projection='3d')
        ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                        cmap='viridis', edgecolor='none')
        ax.set_title(
            'Ackley function in 2D [{}, {}]'.format(-lim, lim))
        ax.set_xlabel('x1')
        ax.set_ylabel('x2')
        ax.set_zlabel('ackley(x1, x2)')
        plt.tight_layout()

    generate_graph(32.768)
    plt.savefig('ackley.png')
    generate_graph(1)
    plt.savefig('ackley2.png')
```

B IMPLEMENTACIÓN DEL ALGORITMO DE ENJAMBRE DE PARTÍCULAS

```python
from typing import Callable, Tuple

import numpy as np
import matplotlib.pyplot as plt


class ParticleSwarmOptimization:
    def __init__(self, eval_fun: Callable[[np.ndarray], np.ndarray],
                 eval_range: Tuple[float, float], n_particles: int = 500,
                 n_iterations: int = 100, alpha: float = 2, beta: float = 2,
                 v_max: float = 2, seed: int = None):
        self.eval = eval_fun
        self.eval_range = eval_range
        self.n_particles = n_particles
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.v_max = v_max
        self.rng = np.random.default_rng(seed)

    def run(self):
        a, b = self.eval_range
        self.x = (b - a) * self.rng.random((self.n_particles, 2)) + a
        local_best = self.x.copy()
        local_evals = self.eval(local_best)
        self.v = np.zeros((self.n_particles, 2))

        global_best_history = np.ndarray((self.n_iterations, 2))

        for i in range(self.n_iterations):
            # Evaluate points
            evals = self.eval(self.x)
            # Update the global best
            global_best = self.x[evals.argmin()]
            global_best_history[i] = global_best
            # Update the local bests
            local_upd_idxs = np.argmin(np.c_[local_evals, evals], axis=1)
            local_upd_idxs = local_upd_idxs.astype(bool)
            local_upd_mask = np.c_[local_upd_idxs, local_upd_idxs]
            local_best[local_upd_mask] = self.x[local_upd_mask]
            local_evals[local_upd_idxs] = evals[local_upd_idxs]
            # Update velocity with global and local bests
            global_v_weight = self.alpha * self.rng.random(2)
            global_v_term = global_v_weight * (global_best - self.x)
            local_v_weight = self.beta * self.rng.random(2)
            local_v_term = local_v_weight * (local_best - self.x)
            self.v += global_v_term + local_v_term
            # Clip velocities greater than v_max
            v_mag = np.linalg.norm(self.v, axis=1).clip(min=self.v_max)
            self.v = self.v_max * self.v / np.c_[v_mag, v_mag]
            # Update positions
            self.x += self.v

        return global_best_history
```

```
    def graph_particle_history(self, particle_history: np.ndarray):
        eval_history = self.eval(particle_history)
        color_progression = np.linspace(1, 0, particle_history.shape[0])

        plt.figure(figsize=(6, 5), dpi=160)
        ax = plt.axes(projection='3d')
        plt.set_cmap('seismic')
        ax.scatter3D(particle_history[:, 0], particle_history[:, 1],
                     eval_history, c=color_progression)
        ax.set_title('Best particle history')
        ax.set_xlabel('x1')
        ax.set_ylabel('x2')
        ax.set_zlabel('ackley(x1, x2)')
        ax.set_xlim(-1, 1)
        ax.set_ylim(-1, 1)
        plt.tight_layout()

    def graph_evaluation_history(self, particle_history: np.ndarray):
        history_size = particle_history.shape[0]
        eval_history = self.eval(particle_history)
        monotonic_eval_history = np.minimum.accumulate(eval_history)

        plt.figure(dpi=160)
        plt.set_cmap('seismic')
        plt.title('Best evaluation curve')
        plt.plot(np.arange(history_size), eval_history, marker='.', ls='')
        plt.plot(np.arange(history_size), monotonic_eval_history,
                 color='green', marker=None)
        plt.yscale('log')
        plt.xlabel('Iterations')
        plt.ylabel('Best evaluation')
        plt.legend(['Evaluation history through iterations',
                    'Best evaluation through iterations'])


if __name__ == "__main__":
    from ackley import ackley_fun

    pso = ParticleSwarmOptimization(ackley_fun, (-32.768, 32.768), seed=0)
    best_particle_history = pso.run()
    pso.graph_particle_history(best_particle_history)
    plt.savefig('pso_particle.png')
    pso.graph_evaluation_history(best_particle_history)
    plt.savefig('pso_evaluation.png')
    best_evaluation_history = ackley_fun(best_particle_history)
    best_idx = np.argmin(best_evaluation_history)
    best_particle = best_particle_history[best_idx]
    best_evaluation = best_evaluation_history[best_idx]
    print('Best particle found on iteration {}: [{:.6f}, {:.6f}] with an evaluation of {:.6f}'.format(
        best_idx+1, best_particle[0], best_particle[1], best_evaluation))
```