

Tesis Proyecto Titulo 1 Particle Swarm Optimization(PSO)

Profesor: Milan Stehlik

Alumno: Fernando Montenegro Zubieta

Contenido

[Introducción: 3](#_Toc144744583)

[Objetivos: 4](#_Toc144744584)

[Objetivo General: 4](#_Toc144744585)

[Objetivo específico: 4](#_Toc144744586)

[Marco metodológico 5](#_Toc144744587)

[Códigos implementados en Python (función 1): 5](#_Toc144744588)

[Librerías utilizadas: 5](#_Toc144744589)

[PSO\_Fishers\_MAX.py 5](#_Toc144744590)

[PSO\_Fishers.py: 7](#_Toc144744591)

[PSO\_Min\_Max.py: 10](#_Toc144744592)

[Códigos implementados en Python (función 2): 11](#_Toc144744593)

[Problemas en desarrollo de proyecto 14](#_Toc144744594)

[Resultados 14](#_Toc144744595)

[Función 1 14](#_Toc144744596)

[PSO\_fishers.py (búsqueda de mínimo) 15](#_Toc144744597)

[PSO\_Fisher\_Max.py (búsqueda de máximo) 16](#_Toc144744598)

[Función 2 17](#_Toc144744599)

[PSO\_ Simple: (maximización) 18](#_Toc144744600)

[PSO\_simple(test de estrés / maximización): 19](#_Toc144744601)

[Optimización PSO 20](#_Toc144744602)

[Double exponential Particle swarm optimization algorithm DExPSO 20](#_Toc144744603)

[Diferencias PSO y DExPSO 20](#_Toc144744604)

[Implementación código DExPSO 21](#_Toc144744605)

[Comparación de tiempo de búsqueda de punto de convergencia 21](#_Toc144744606)

[PSO 21](#_Toc144744607)

[DExPSO 21](#_Toc144744608)

[Conclusión de comparación 21](#_Toc144744609)

[Cronograma del proyecto: 22](#_Toc144744610)

[Bibliografía 22](#_Toc144744611)

# Introducción:

El algoritmo de Optimización por Enjambre de Partículas (PSO) es una técnica de optimización inspirada en el comportamiento social de los pájaros o peces. En PSO, un grupo de partículas (o agentes) se mueven en el espacio de búsqueda para encontrar la solución óptima a un problema dado. Cada partícula representa una posible solución al problema y se mueve en función de su propia experiencia y la de las mejores partículas en el enjambre. PSO es simple pero poderoso y puede usarse para resolver problemas de optimización como selección de características, ajuste de parámetros y optimización de funciones.

“PSO (Particle Swarm Optimization) es una metaheurística en el campo de los sistemas inteligentes que permite encontrar el valor óptimo de una función objetivo determinada. Para ello utiliza un conjunto de partículas que se desplazan y colaboran entre sí en un espacio n-dimensional. Este proceso demanda un costo computacional que depende de la función analizada y de la estrategia empleada” (Miguel A. Azar, Fabiola P. Paz, Analía Herrera, p. 1)

[Universidad Carlos III Madrid: Particle Swarm Intelligence](http://tracer.uc3m.es/tws/pso/) (2019)

Cada partícula (individuo) tiene una **posición**, *p*⃗  (que en 2 dimensiones vendrá determinado por un vector de la forma (*x*,*y*)), en el espacio de búsqueda y una **velocidad**, *v*⃗  (que en 2 dimensiones vendrá determinado por un vector de la forma (*vx*,*vy*)), que determina su movimiento a través del espacio. Además, como partículas de un mundo real físico, tienen una cantidad de **inercia**, que los mantiene en la misma dirección en la que se movían, así como una aceleración (cambio de velocidad), que depende principalmente de dos características:

1. Cada partícula es atraída hacia la mejor localización que ella, personalmente, ha encontrado en su historia (**mejor personal**).
2. Cada partícula es atraída hacia la mejor localización que ha sido encontrada por el conjunto de partículas en el espacio de búsqueda (**mejor global**) (p. 1).

Imagen1:

Fuerzas aplicadas a una partícula en PSO:

Diagrama

Descripción generada automáticamente

[Universidad Carlos III Madrid: Particle Swarm Intelligence](http://tracer.uc3m.es/tws/pso/) (2019)

# Objetivos:

Objetivo General: Optimización de función en algoritmo PSO. El objetivo de la optimización es encontrar la posición de la partícula que produce el mejor valor de la función de costo o punto de convergencia (ya sea un máximo o un mínimo).

Objetivo específico: Esta implementación debe llevarse a cabo en Python y encontrar un mínimo y máximo de la función.

Las funciones serán continuas y semicontinuas.

Se debe comparar con resultados conocidos en relación el tiempo de ejecución y resultados.

# Marco metodológico

El objetivo de esta investigación es la comparación de los resultados, eficiencia y efectividad de aplicar Particle swarm optimization(PSO) en Python con funciones complejas. Para esta implantación la población por defecto para cada estudio de una función será de 20 partículas y 1000 repeticiones para que la partícula encuentre su mejor posición y punto de convergencia, las inercia(W) para casos de esta implementación será de 0.729, el peso cognitivo de 1.49445 y peso social será de 1.49445.

Los instrumentos usados en esta investigación son:

1.- El estudio “Some Properties of D-optimal Designs for Random Fields with Different Variograms”

2.- El editor de Texto Visual studio code.

3.- Procesador Intel i5-10600kf.

# Códigos implementados en Python (función 1):

## Librerías utilizadas:

* Numpy as np: La librería **NumPy**, cuando se importa como np, se utiliza en el algoritmo PSO principalmente para realizar cálculos numéricos eficientes y manipular matrices y vectores multidimensionales.

NumPy incluye una amplia gama de funciones matemáticas que pueden ser útiles en la evaluación de la función objetivo y otros cálculos en PSO.

* La librería **math** se utiliza en PSO para realizar cálculos matemáticos precisos, evaluar la función objetivo y realizar operaciones matemáticas esenciales. Las funciones y constantes proporcionadas por math pueden ser utilizadas para definir y manipular datos matemáticos en el contexto del algoritmo PSO.
* La librería **Time** para casos de este estudio solo se utiliza para saber el tiempo de ejecución del programa y conocer la eficiencia a partir de ese tiempo.

## PSO\_Fishers\_MAX.py

import numpy as np

import math

import time

# Definir la función objetivo

def objective\_function(x):

    y = x

    z = x

    v = x

    return (-5 - 2 \* math.exp(-x-2\*y-2\*z-2\*v) - 2 \* math.exp(-2\*x-2\*y-z-2\*v) + 2 \* math.exp(-z-2\*v-2\*x) - 2 \* math.exp(-2\*x-2\*z-2\*v-y) - 2 \* math.exp(-z-2\*v) + 2 \* math.exp(-2\*y-z-2\*v) + 2 \* math.exp(-2\*v-y-2\*x) + 2 \* math.exp(-2\*y-2\*z-v) - 2 \* math.exp(-2\*x-2\*y-2\*z-v) + 2 \* math.exp(-2\*y-x-2\*v) + 2 \* math.exp(-2\*z-v-2\*x) + 3 \* math.exp(-x) + 2 \* math.exp(-x) - 2 \* math.exp(-v-2\*y) + 2 \* math.exp(-2\*y-2\*x-v) + 2 \* math.exp(-2\*z-2\*v-y) - 2 \* math.exp(-v-2\*x) - math.exp(-y-2\*v) - math.exp(-2\*z-2\*v) - 2 \* math.exp(-2\*z-v) - math.exp(-2\*y-2\*z-2\*v) + 3 \* math.exp(-2\*v) - math.exp(-2\*v-2\*y) + 2 \* math.exp(-2\*y-x-2\*z) + 2 \* math.exp(-z) + 2 \* math.exp(1) - y - 2 \* math.exp(-2\*y-x) - 2 \* math.exp(-x-2\*z) + 2 \* math.exp(-2\*z-2\*x-y) - 2 \* math.exp(-y-2\*z) - 2 \* math.exp(-2\*x-y) + 3 \* math.exp(-2\*z) + 3 \* math.exp(-2\*y) - 2 \* math.exp(-2\*y-z) - 2 \* math.exp(-2\*v-x) - math.exp(-2\*y-2\*z) + 2 \* math.exp(-2\*x-2\*y-z)+ 2 \* math.exp(x-2\*z-2\*v) - 2 \* math.exp(-v) - math.exp(-z-2\*x) - math.exp(-2\*x-2\*z) - math.exp(-2\*x-2\*y) +  math.exp(-2\*x-2\*y-2\*z) - 3 \* math.exp(-2\*x-2\*y-2\*z-2\*v) - math.exp(-2\*z-2\*v-2\*x) - math.exp(-2\*y-2\*x-2\*v) - math.exp(-2\*v-2\*x)) / (-1 + math.exp(-2\*x) - math.exp(-2\*z-2\*v) + math.exp(-2\*y-2\*z-2\*v) + math.exp(-2\*v) - math.exp(-2\*v-2\*y) + math.exp(-2\*z) + math.exp(-2\*y) - math.exp(-2\*y-2\*z) - math.exp(-2\*x-2\*z) + math.exp(-2\*x-2\*y) + math.exp(-2\*x-2\*y-2\*z) - math.exp(-2\*x-2\*y-2\*z-2\*v) + math.exp(-2\*z-2\*v-2\*x) + math.exp(-2\*y-2\*x-2\*v) - math.exp(-2\*v-2\*x))

class Particle:

    def \_\_init\_\_(self, dim, values):

        self.position = np.array([np.random.choice(values) for \_ in range(dim)]) # se escoje un valor al azar de la lista.

        self.velocity = np.zeros(dim) # Inicializa la velocidad de la partícula como un vector de ceros de la misma dimensión que la posición.

        self.best\_position = self.position.copy() # Inicializa la mejor posición de la partícula como su posición actual

        self.best\_fitness = float('-inf') # Inicializa la mejor aptitud de la partícula como infinito positivo.

                                         # Esta es una estrategia común para asegurarse de que cualquier valor de aptitud real sea menor y, por lo tanto, se actualizará en la primera iteración.

    def update\_best(self, function):

        fitness = function(self.position) # Calcula la aptitud de la partícula actual utilizando la función de aptitud dada como argumento.

        if fitness > self.best\_fitness:

            self.best\_fitness = fitness

            self.best\_position = self.position.copy()

class PSO:

    def \_\_init\_\_(self, function, dim, size, values, iterations):

        self.function = function

        self.dim = dim

        self.swarm = [Particle(dim, values) for \_ in range(size)]

        self.global\_best\_position = np.array([np.random.choice(values) for \_ in range(dim)])

        self.global\_best\_fitness = float('inf')

        self.iterations = iterations

    def run(self):

        start\_time = time.time()  # Guarda el tiempo de inicio

        for i in range(self.iterations):

            for particle in self.swarm:

                particle.update\_best(self.function)

                if particle.best\_fitness < self.global\_best\_fitness:

                    self.global\_best\_fitness = particle.best\_fitness

                    self.global\_best\_position = particle.best\_position.copy()

                w = 0.729  # Inertial weight

                c1 = 1.49445  # Cognitive weight

                c2 = 1.49445  # Social weight

                r1 = np.random.rand(self.dim)

                r2 = np.random.rand(self.dim)

                particle.velocity = (w \* particle.velocity) + (c1 \* r1 \* (particle.best\_position - particle.position)) + (c2 \* r2 \* (self.global\_best\_position - particle.position))

                particle.position += particle.velocity

                particle.position = np.array([np.clip(val, min(values), max(values)) for val in particle.position])

        end\_time = time.time()  # Guarda el tiempo de fin

        elapsed\_time = end\_time - start\_time  # Calcula el tiempo transcurrido

        return self.global\_best\_position, self.global\_best\_fitness, elapsed\_time

values = [1.0]  # Valores posibles para x

pso = PSO(lambda x: -objective\_function(x), 1, 20, values, 1000)

best\_max\_position, best\_max\_fitness, elapsed\_time = pso.run()

print("Mejor posición máxima encontrada:", best\_max\_position) # posición en la cual se obtiene el valor máximo de la función.

print("Valor de función máxima encontrado:", -best\_max\_fitness)

print("Tiempo de ejecución:", elapsed\_time, "segundos")

Este código es el primero que se aplicó una comparación con el estudio mencionado anteriormente(Some Properties…). La intención es que encuentre un máximo de función con los valores dados en la lista (values). Para ello el mejor valor de función siempre inicia en -infinito lo que garantiza que el valor encontrado en el primer ciclo sea el nuevo mejor valor de la función el cual ira cambiando a medida que el código avanza y encuentre mejores resultados.

## PSO\_Fishers.py:

import numpy as np

import math

import time

# Definir la función objetivo

def objective\_function(x):

    y = x

    z = x

    v = x

    return (-5 - 2 \* math.exp(-x-2\*y-2\*z-2\*v) - 2 \* math.exp(-2\*x-2\*y-z-2\*v) + 2 \* math.exp(-z-2\*v-2\*x) - 2 \* math.exp(-2\*x-2\*z-2\*v-y) - 2 \* math.exp(-z-2\*v) + 2 \* math.exp(-2\*y-z-2\*v) + 2 \* math.exp(-2\*v-y-2\*x) + 2 \* math.exp(-2\*y-2\*z-v) - 2 \* math.exp(-2\*x-2\*y-2\*z-v) + 2 \* math.exp(-2\*y-x-2\*v) + 2 \* math.exp(-2\*z-v-2\*x) + 3 \* math.exp(-x) + 2 \* math.exp(-x) - 2 \* math.exp(-v-2\*y) + 2 \* math.exp(-2\*y-2\*x-v) + 2 \* math.exp(-2\*z-2\*v-y) - 2 \* math.exp(-v-2\*x) - math.exp(-y-2\*v) - math.exp(-2\*z-2\*v) - 2 \* math.exp(-2\*z-v) - math.exp(-2\*y-2\*z-2\*v) + 3 \* math.exp(-2\*v) - math.exp(-2\*v-2\*y) + 2 \* math.exp(-2\*y-x-2\*z) + 2 \* math.exp(-z) + 2 \* math.exp(1) - y - 2 \* math.exp(-2\*y-x) - 2 \* math.exp(-x-2\*z) + 2 \* math.exp(-2\*z-2\*x-y) - 2 \* math.exp(-y-2\*z) - 2 \* math.exp(-2\*x-y) + 3 \* math.exp(-2\*z) + 3 \* math.exp(-2\*y) - 2 \* math.exp(-2\*y-z) - 2 \* math.exp(-2\*v-x) - math.exp(-2\*y-2\*z) + 2 \* math.exp(-2\*x-2\*y-z)+ 2 \* math.exp(x-2\*z-2\*v) - 2 \* math.exp(-v) - math.exp(-z-2\*x) - math.exp(-2\*x-2\*z) - math.exp(-2\*x-2\*y) +  math.exp(-2\*x-2\*y-2\*z) - 3 \* math.exp(-2\*x-2\*y-2\*z-2\*v) - math.exp(-2\*z-2\*v-2\*x) - math.exp(-2\*y-2\*x-2\*v) - math.exp(-2\*v-2\*x)) / (-1 + math.exp(-2\*x) - math.exp(-2\*z-2\*v) + math.exp(-2\*y-2\*z-2\*v) + math.exp(-2\*v) - math.exp(-2\*v-2\*y) + math.exp(-2\*z) + math.exp(-2\*y) - math.exp(-2\*y-2\*z) - math.exp(-2\*x-2\*z) + math.exp(-2\*x-2\*y) + math.exp(-2\*x-2\*y-2\*z) - math.exp(-2\*x-2\*y-2\*z-2\*v) + math.exp(-2\*z-2\*v-2\*x) + math.exp(-2\*y-2\*x-2\*v) - math.exp(-2\*v-2\*x))

class Particle:

    def \_\_init\_\_(self, dim, values):

        self.position = np.array([np.random.choice(values) for \_ in range(dim)]) # se escoje un valor al azar de la lista.

        self.velocity = np.zeros(dim) # Inicializa la velocidad de la partícula como un vector de ceros de la misma dimensión que la posición.

        self.best\_position = self.position.copy() # Inicializa la mejor posición de la partícula como su posición actual

        self.best\_fitness = float('inf') # Inicializa la mejor aptitud de la partícula como infinito positivo.

                                         # Esta es una estrategia común para asegurarse de que cualquier valor de aptitud real sea menor y, por lo tanto, se actualizará en la primera iteración.

    def update\_best(self, function):

        fitness = function(self.position) # Calcula la aptitud de la partícula actual utilizando la función de aptitud dada como argumento.

        if fitness < self.best\_fitness:

            self.best\_fitness = fitness

            self.best\_position = self.position.copy()

class PSO:

    def \_\_init\_\_(self, function, dim, size, values, iterations):

        self.function = function

        self.dim = dim

        self.swarm = [Particle(dim, values) for \_ in range(size)]

        self.global\_best\_position = np.array([np.random.choice(values) for \_ in range(dim)])

        self.global\_best\_fitness = float('inf')

        self.iterations = iterations

    def run(self):

        start\_time = time.time()  # Guarda el tiempo de inicio

        for i in range(self.iterations):

            for particle in self.swarm:

                particle.update\_best(self.function)

                if particle.best\_fitness < self.global\_best\_fitness:

                    self.global\_best\_fitness = particle.best\_fitness

                    self.global\_best\_position = particle.best\_position.copy()

                w = 0.729  # Inertial weight

                c1 = 1.49445  # Cognitive weight

                c2 = 1.49445  # Social weight

                r1 = np.random.rand(self.dim)

                r2 = np.random.rand(self.dim)

                particle.velocity = (w \* particle.velocity) + (c1 \* r1 \* (particle.best\_position - particle.position)) + (c2 \* r2 \* (self.global\_best\_position - particle.position))

                particle.position += particle.velocity

                particle.position = np.array([np.clip(val, min(values), max(values)) for val in particle.position])

        end\_time = time.time()  # Guarda el tiempo de fin

        elapsed\_time = end\_time - start\_time  # Calcula el tiempo transcurrido

        return self.global\_best\_position, self.global\_best\_fitness, elapsed\_time

# Ejemplo de uso

values = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.0, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0]  # Valores posibles para x

pso = PSO(objective\_function, 1, 20, values, 1000)

best\_min\_position, best\_min\_fitness, elapsed\_time = pso.run()

print("Mejor posición mínima encontrada:", best\_min\_position)

print("Valor de función mínima encontrado:", best\_min\_fitness) # posición en la cual se obtiene el valor mínimo de la función.

print("Tiempo de ejecución:", elapsed\_time, "segundos")

En este código a diferencia con el anterior busca el mínimo de la función. Para ello usa la misma estrategia que el anterior con la diferencia que la función parte en infinito lo que garantiza que en el primer ciclo se cambie el valor por un mejor resultado, el cual puede seguir cambiando a medida avanza el código en su ejecución.

## PSO\_Min\_Max.py:

import numpy as np

# Definir la función objetivo

def objective\_function(x):

    return (x\*\*2 - 1)\*\*2

class Particle:

    def \_\_init\_\_(self, dim, minx, maxx):

        self.position = np.random.uniform(low=minx, high=maxx, size=dim)

        self.velocity = np.zeros(dim)

        self.best\_position = self.position.copy()

        self.best\_fitness = float('inf')

    def update\_best(self, function):

        fitness = function(self.position)

        if fitness < self.best\_fitness:

            self.best\_fitness = fitness

            self.best\_position = self.position.copy()

class PSO:

    def \_\_init\_\_(self, function, dim, size, minx, maxx, iterations):

        self.function = function

        self.dim = dim

        self.swarm = [Particle(dim, minx, maxx) for \_ in range(size)]

        self.global\_best\_position = np.random.uniform(low=minx, high=maxx, size=dim)

        self.global\_best\_fitness = float('inf')

        self.iterations = iterations

    def run(self):

        for i in range(self.iterations):

            for particle in self.swarm: #es la población de partículas que se utilizan para explorar el espacio de búsqueda y encontrar la mejor solución a la función objetivo.

                particle.update\_best(self.function)

                if particle.best\_fitness < self.global\_best\_fitness:

                    self.global\_best\_fitness = particle.best\_fitness

                    self.global\_best\_position = particle.best\_position.copy()

                w = 0.729  # Inertial weight

                c1 = 1.49445  # Cognitive weight

                c2 = 1.49445  # Social weight

                r1 = np.random.rand(self.dim)

                r2 = np.random.rand(self.dim)

                particle.velocity = (w \* particle.velocity) + (c1 \* r1 \* (particle.best\_position - particle.position)) + (c2 \* r2 \* (self.global\_best\_position - particle.position))

                particle.position += particle.velocity

                particle.position = np.clip(particle.position, -100, 100)

        return self.global\_best\_position, self.global\_best\_fitness

# Ejemplo de uso

pso = PSO(objective\_function, 1, 5, -10, 10, 100)

best\_min\_position, best\_min\_fitness = pso.run()

print("Mejor posición mínima encontrada:", best\_min\_position)

print("Mejor valor de aptitud mínima encontrado:", best\_min\_fitness)

pso = PSO(lambda x: -objective\_function(x), 1, 5, -10, 10, 100)

best\_max\_position, best\_max\_fitness = pso.run()

print("Mejor posición máxima encontrada:", best\_max\_position)

print("Mejor valor de aptitud máxima encontrado:", -best\_max\_fitness)

En este código fue el primer paso para implementar PSO con una función más simple (X\*\*2 - 1)\*\*2 y además de que cuenta con una población menor y menor cantidad de repeticiones para cada partícula en buscar su mejor mínimo o máximo.

# Códigos implementados en Python (función 2):

import numpy as np

import math

import time

def objective\_function(x):

    y = x

    z = x

    return 2\*(-2 + math.exp(-z) + math.exp(-x) + math.exp(-y) + math.exp(-2\*y-x-2\*z) + math.exp(-2\*x) + math.exp(-2\*y) + math.exp(-2\*z) - math.exp(-2\*x-2\*y-2\*z) - math.exp(-x -2\*z) + math.exp(-2\*x -2\*y -z) - math.exp(-z -2\*x) - math.exp(-y -2\*z) -math.exp(-2\*x -y) - math.exp(-2\*y -z) + math.exp(-2\*z -2\*x -y) - math.exp(-2\*y -x))/(-1 + math.exp(-2\*z) + math.exp(-2\*y)- math.exp(-2\*y -2\*z) + math.exp(-2\*x) - math.exp(-2\*x -2\*z) - math.exp(-2\*x -2\*y) + math.exp(-2\*x -2\*y -2\*z))

class Particle:

    def \_\_init\_\_(self, dim, values):

        self.position = np.full(dim, values) #

        self.velocity = np.zeros(dim) # Inicializa la velocidad de la partícula como un vector de ceros de la misma dimensión que la posición.

        self.best\_position = self.position.copy() # Inicializa la mejor posición de la partícula como su posición actual

        self.best\_fitness = float('-inf') # Inicializa la mejor aptitud de la partícula como infinito positivo.

                                         # Esta es una estrategia común para asegurarse de que cualquier valor de aptitud real sea menor y, por lo tanto, se actualizará en la primera iteración.

    def update\_best(self, function):

        fitness = function(self.position) # Calcula la aptitud de la partícula actual utilizando la función de aptitud dada como argumento.

        if fitness > self.best\_fitness:

            self.best\_fitness = fitness

            self.best\_position = self.position.copy()

class PSO:

    def \_\_init\_\_(self, function, dim, size, values, iterations):

        self.function = function

        self.dim = dim

        self.swarm = [Particle(dim, values) for \_ in range(size)]

        self.global\_best\_position = values

        self.global\_best\_fitness = float('inf')

        self.iterations = iterations

    def run(self):

        #start\_time = time.time()  # Guarda el tiempo de inicio

        for i in range(self.iterations):

            for particle in self.swarm:

                particle.update\_best(self.function)

                if particle.best\_fitness < self.global\_best\_fitness:

                    self.global\_best\_fitness = particle.best\_fitness

                    self.global\_best\_position = particle.best\_position.copy()

                w = 0.729  # Inertial weight

                c1 = 1.49445  # Cognitive weight

                c2 = 1.49445  # Social weight

                if self.dim > 1:

                    r1 = 1

                    r2 = 1

                else:

                    r1 = 1

                    r2 = 1

                particle.velocity = (w \* particle.velocity) + (c1 \* r1 \* (particle.best\_position - particle.position)) + (c2 \* r2 \* (self.global\_best\_position - particle.position))

                particle.position += particle.velocity

                particle.position = np.clip(particle.position, values, values)

        return self.global\_best\_position, self.global\_best\_fitness

start\_time = time.time()  # Guarda el tiempo de inicio

values = 0.1

convergence\_point = 0

while True:

    pso = PSO(lambda x: -objective\_function(x), 1, 20, values, 1000)

    best\_max\_position, best\_max\_fitness = pso.run()

    print("Mejor posición máxima encontrada:", best\_max\_position)

    print("Valor de función máxima encontrado:", -best\_max\_fitness)

    if convergence\_point < -best\_max\_fitness :

        convergence\_point = -best\_max\_fitness

    if values > 10:

        break

    else:

        values += 0.1

print("Punto de convergencia: ",convergence\_point)

end\_time = time.time()  # Guarda el tiempo de fin

elapsed\_time = end\_time - start\_time  # Calcula el tiempo transcurrido

print("Tiempo de ejecución:", elapsed\_time, "segundos")

A diferencia de la implementación de la primera función este código permite ejecutar continuamente la función dentro del rango que se le dé. El cual será aplicado a cada código posterior para agilizar su evaluación de resultados.

# Problemas en desarrollo de proyecto

Unos de los problemas iniciales de este código fue la gran función a evaluar en las primeras ejecuciones los resultados eran demasiado disparejos por lo que se le dio una revisión más exhausta y la función no estaba totalmente igual al del estudio que se debe comparar. Además el programa no estaba optimizado para un mínimo o un máximo en la misma ejecución, por lo que se decidió separar estos procesos para que el programa a solo busque un mínimo o un máximo no ambos.

# Resultados

## Función 1

Función:

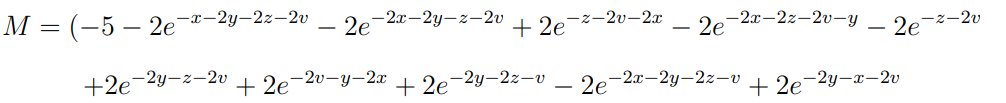


Imagen en blanco y negro

Descripción generada automáticamente con confianza media

### PSO\_fishers.py (búsqueda de mínimo)

Tabla

Descripción generada automáticamente

En la imagen anterior se puede observar desde que el valor de la partícula o “X” es igual a 0.6 existe un error, ya que, la función que se esta evaluando no es negativa. Por observación y estudios en este problema se pudo identificar que unos de los problemas era la función que se evaluaba.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

### PSO\_Fisher\_Max.py (búsqueda de máximo)

Tabla

Descripción generada automáticamente

De igual forma que los datos de la búsqueda anterior estos presentan un error cuando la partícula o “X” obtiene un valor igual a 0,6.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Datos obtenidos al ejecutar la implementación de pso en esta primera función a evaluar tanto al minimizar y maximizar respectivamente.

## Función 2

Función:

Gráfico, Gráfico de cajas y bigotes

Descripción generada automáticamente

### PSO\_ Simple: (maximización)

Gráfico, Gráfico de líneas, Gráfico de dispersión

Descripción generada automáticamente

En la imagen anterior vemos los resultados obtenidos de aplicar PSO en un rango [0,1] para este caso nuestro punto de convergencia es: 2,386 con un tiempo de ejecución de 12,154 segundos. Hay que tener en cuenta que el punto de convergencia poder aumentar si a esa misma función se le aplica un test de estrés como se mostrara a continuación.

### PSO\_simple(test de estrés / maximización):

Gráfico, Gráfico de líneas

Descripción generada automáticamente

En esta segunda función al aplicar PSO y hacer un test de estrés los resultados que obtenemos del grafico anterior siendo su punto de convergencia el valor: 3,963 en el rango [0,5] y un tiempo de ejecución de: 55,976 segundos que se considera óptima para esta implementación. Si hacemos una prueba de estrés al mismo algoritmo en un rango [0,10] el valor de convergencia queda en 3,999 y un tiempo de ejecución de: 110,705 segundos.

# Optimización PSO

## Double exponential Particle swarm optimization algorithm DExPSO

“La idea clave aquí es aproximar la variable aleatoria uniforme mediante otra variable aleatoria con soporte en toda la recta real, como en un proceso de suavizado. Estos requisitos garantizan características deseables, como que las propiedades de convergencia del algoritmo se transferirán al nuevo algoritmo. (Milan Stehlík, Weng Kee Wong , Ping-Yang Chen and Jozef Kiseľák, p. 8)

Según GPT 3.5: “En DExPSO, las partículas actualizan sus velocidades en función de una distribución de doble exponencial. Esta distribución ayuda a controlar el equilibrio entre la exploración y la explotación en el espacio de búsqueda, mejorando la capacidad del algoritmo para escapar de los óptimos locales y explorar nuevas regiones.”

## Diferencias PSO y DExPSO

Una de las principales diferencias a recalcar entre estos 2 métodos son que PSO puede quedar atrapado en un óptimo local y no encontrar el óptimo global debido al uso de variables aleatorias uniformes. Mientras que DExPSO usa variables aleatorias de una distribución no uniforme lo que puede ser más eficiente para no quedar atrapado en un óptimo local.

Mecanismo de actualización:

1. PSO: En PSO, las partículas se actualizan basándose en su mejor posición personal, la mejor posición global del enjambre y una componente aleatoria. La actualización se realiza mediante una fórmula que combina estas influencias.
2. DExPSO: Utiliza una combinación de PSO y el algoritmo de Evolución Diferencial (DE). En lugar de utilizar una fórmula simple de actualización, DExPSO utiliza operadores de mutación y recombinación inspirados en DE para guiar las actualizaciones de las partículas.

Convergencia y eficiencia:

1. **PSO**: Puede requerir ajustes cuidadosos de parámetros para lograr una convergencia efectiva y evitar la convergencia prematura.
2. **DExPSO**: Debido a su enfoque en la explotación guiada por DE, podría ser más eficaz en ciertos problemas y requerir menos ajustes de parámetros.

Según lo leído en articulo [[4]](#_Bibliografía) de la pagina 13 podemos ver una ejemplo de comparación que se compara PSO y DExPSO con la función Rastrigin que se encarga de comprobar la optimización de los algoritmos, en la siguiente tabla se muestran los resultados de la comparación.

Un reloj de aguja

Descripción generada automáticamente con confianza media

“Table 1 displays the results and clearly shows that PSO with uniform variates has less optimal results than the DExPSO algorithm where the stochastic components have the DEx density. We also observe, that for a fixed number of particles, PSO got stuck in some local extrema whereas the proposed version of PSO did not. On the other hand, increasing the number of particles does not solve the situation, it may even worsen the result. The latter theoretically guarantees some kind of convergence of our proposed procedure.”

Tabla

Descripción generada automáticamente

# Implementación código DExPSO

Según articulo [[5]](#_Bibliografía) pagina 8 para esta aplicación de DExPSO y en busca de una mejor optimización el peso cambia con el tiempo de manera lineal para mejorar la exploración global y la exploración local en diferentes etapas de la búsqueda.

import numpy as np

import math

import time

class Particle:

    def \_\_init\_\_(self, dim, values):

        self.position = np.random.uniform(values, values, dim)

        self.velocity = np.zeros(dim)

        self.best\_position = self.position.copy()

        self.best\_fitness = float('-inf')

    def update\_best(self, function):

        fitness = function(self.position)

        if fitness > self.best\_fitness:

            self.best\_fitness = fitness

            self.best\_position = self.position.copy()

class DExPSO:

    def \_\_init\_\_(self, function, dim, size, values, iterations):

        self.function = function

        self.dim = dim

        self.swarm = [Particle(dim, values) for \_ in range(size)]

        self.global\_best\_position = np.random.uniform(values, values, dim)

        self.global\_best\_fitness = float('-inf')

        self.iterations = iterations

    def run(self):

        for i in range(self.iterations):

            for particle in self.swarm:

                particle.update\_best(self.function)

                if particle.best\_fitness > self.global\_best\_fitness:

                    self.global\_best\_fitness = particle.best\_fitness

                    self.global\_best\_position = particle.best\_position.copy()

                c1 = 1.49445

                c2 = 1.49445

                w = np.exp(-i / self.iterations)  # Exponential decay of inertia weight

                for d in range(self.dim):

                    r1 = np.random.rand()

                    r2 = np.random.rand()

                    s = -1 if np.random.rand() < 0.5 else 1

                    particle.velocity[d] = w \* particle.velocity[d] + c1 \* r1 \* (particle.best\_position[d] - particle.position[d]) + c2 \* r2 \* (self.global\_best\_position[d] - particle.position[d])

                    particle.position[d] += s \* np.exp(-i / self.iterations) \* particle.velocity[d]

        return self.global\_best\_position, self.global\_best\_fitness

def objective\_function(x):

    y = x

    z = x

    return 2\*(-2 + math.exp(-z) + math.exp(-x) + math.exp(-y) + math.exp(-2\*y-x-2\*z) + math.exp(-2\*x) + math.exp(-2\*y) + math.exp(-2\*z) - math.exp(-2\*x-2\*y-2\*z) - math.exp(-x -2\*z) + math.exp(-2\*x -2\*y -z) - math.exp(-z -2\*x) - math.exp(-y -2\*z) -math.exp(-2\*x -y) - math.exp(-2\*y -z) + math.exp(-2\*z -2\*x -y) - math.exp(-2\*y -x))/(-1 + math.exp(-2\*z) + math.exp(-2\*y)- math.exp(-2\*y -2\*z) + math.exp(-2\*x) - math.exp(-2\*x -2\*z) - math.exp(-2\*x -2\*y) + math.exp(-2\*x -2\*y -2\*z))

# Parámetros del algoritmo DExPSO

dim = 1

values = 0.1

size = 20

iterations = 1000

start\_time = time.time()

convergence\_point = float('-inf')  # Punto de convergencia inicial

while True:

    pso = DExPSO(lambda x: -objective\_function(x), 1, 20, values, 1000)

    best\_max\_position, best\_max\_fitness = pso.run()

    print("Mejor posición máxima encontrada:", best\_max\_position)

    print("Valor de función máxima encontrado:", -best\_max\_fitness)

    if convergence\_point < -best\_max\_fitness:

        convergence\_point = -best\_max\_fitness

    if values > 10:

        break

    else:

        values += 0.1

print("Punto de convergencia: ", convergence\_point)

end\_time = time.time()  # Guarda el tiempo de fin

elapsed\_time = end\_time - start\_time  # Calcula el tiempo transcurrido

print("Tiempo de ejecución:", elapsed\_time, "segundos")

## Comparación de tiempo de búsqueda de punto de convergencia

En este apartado se mostrará la comparación de los dos algoritmos tanto de PSO como DExPSO para encontrar el punto de convergencia para la función 1 y 2 expuesta anteriormente con un rango de [0,10]. Se comparar el punto de convergencia encontrado y el tiempo de ejecución de ambas partes.

## Función 1:

### PSO

* Punto de convergencia: 4.999671
* Tiempo de ejecución: 308.23947 segundos

### DExPSO

* Punto de convergencia: 4.999671
* Tiempo de ejecución: 275.83437 segundos

## Función 2:

### PSO

* Punto de convergencia: 3.999753
* Tiempo de ejecución: 113.33807 segundos

### DExPSO

* Punto de convergencia: 3.999753
* Tiempo de ejecución: 84.99834 segundos

### Conclusión de comparación

Dado los resultados expuestos anteriormente DExPSO tiene una eficiencia en términos de tiempo de ejecución en la función 1 de un 10,51% sobre PSO y en la función 2 tiene una eficiencia de 25% sobre PSO.

Para seguir comprobando eficiencia se plantea la siguiente pregunta: ¿Para la misma función 1 si cambiamos z por un valor aleatorio entre un rango [2,3] se mantendrá la misma eficiencia o similar?

Para esto z se cambiará por llamado que se la haga a la función:

Texto

Descripción generada automáticamente

* np: Esto es un alias que generalmente se usa para referirse a la biblioteca NumPy. NumPy es una biblioteca en Python que proporciona estructuras de datos eficientes para trabajar con arreglos numéricos y funciones matemáticas.
* np.random: En NumPy, el módulo random proporciona funciones para generar números aleatorios.
* np.random.uniform(2, 3): La función uniform genera números aleatorios con una distribución uniforme en el intervalo [2, 3). En otras palabras, z tomará un valor aleatorio entre 2 (incluido) y 3 (excluido).

Resultados PSO (función 1):

* Punto de convergencia: 4.9043
* Tiempo de ejecución: 289.19057 segundos

Resultados DExPSO (función 1):

* Punto de convergencia: 4.7613
* Tiempo de ejecución: 255.213247 segundos

Si bien es esperable que el punto de convergencia entre ambos programas sea distinto ya que z cambia en cada llamada de función dentro del rango dado, la eficiencia de DExPSO se mantiene sobre PSO en un 11,74% muy similar a la prueba anterior.

# PSO 2 dimensiones

En su versión bidimensional, PSO busca encontrar la solución óptima de un problema considerando dos variables. Imagina un enjambre de partículas en un espacio de búsqueda, donde cada partícula representa una posible solución. Estas partículas se mueven hacia regiones más prometedoras del espacio según su posición y velocidad.

En cada iteración, cada partícula ajusta su velocidad y posición basándose en su experiencia previa y en la mejor posición alcanzada por el enjambre. La fórmula de actualización de la velocidad y posición incorpora componentes de inercia, cognitiva y social, que determinan la dirección del movimiento. Este proceso se repite hasta convergencia o alcanzar un número predefinido de iteraciones.

La implementación en 2 dimensiones expande las capacidades de PSO para abordar problemas más complejos, permitiendo la exploración simultánea de dos variables y mejorando la convergencia hacia soluciones óptimas en el espacio de búsqueda bidimensional.

Según los programas anteriores obtenemos el siguiente código para PSO en 2 dimensiones:

import numpy as np

import math

import time

r = 0.1

class Particle:

    def \_\_init\_\_(self):

        self.position = np.random.uniform(low=0, high=100, size=2)

        self.velocity = np.random.uniform(low=-1.0, high=1.0, size=2)

        self.best\_position = np.copy(self.position)

        self.best\_fitness = float('-inf')

    def update\_best(self, objective\_function):

        fitness = objective\_function(self.position[0], self.position[1])

        if fitness > self.best\_fitness:

            self.best\_fitness = fitness

            self.best\_position = np.copy(self.position)

    def update\_velocity\_and\_position(self, global\_best\_position, w, c1, c2):

        r1 = np.random.rand(2)

        r2 = np.random.rand(2)

        self.velocity = (w \* self.velocity) + (c1 \* r1 \* (self.best\_position - self.position)) + (c2 \* r2 \* (global\_best\_position - self.position))

        self.position += self.velocity

        self.position = np.clip(self.position, -5.0, 5.0)

class PSO:

    def \_\_init\_\_(self, objective\_function, n\_particles, iterations):

        self.objective\_function = objective\_function

        self.swarm = [Particle() for \_ in range(n\_particles)]

        self.global\_best\_position = np.random.uniform(low=0, high=100, size=2)

        self.global\_best\_fitness = float('-inf')

        self.w = 0.5

        self.c1 = 1.5

        self.c2 = 1.5

        self.iterations = iterations

    def optimize(self):

        start\_time = time.time()

        for i in range(self.iterations):

            for particle in self.swarm:

                particle.update\_best(self.objective\_function)

                if particle.best\_fitness > self.global\_best\_fitness:

                    self.global\_best\_fitness = particle.best\_fitness

                    self.global\_best\_position = np.copy(particle.best\_position)

            for particle in self.swarm:

                particle.update\_velocity\_and\_position(self.global\_best\_position, self.w, self.c1, self.c2)

        end\_time = time.time()

        elapsed\_time = end\_time - start\_time

        return self.global\_best\_position, self.global\_best\_fitness, elapsed\_time

# Definir la función de prueba (puedes reemplazarla con tu propia función)

def test\_function(x, y):

while True:

    pso = PSO(test\_function, 20, 1000)

    best\_position, best\_fitness, execution\_time = pso.optimize()

    r = r + 0.1

    # Imprimir resultados

    print("Mejor posición encontrada:", best\_position)

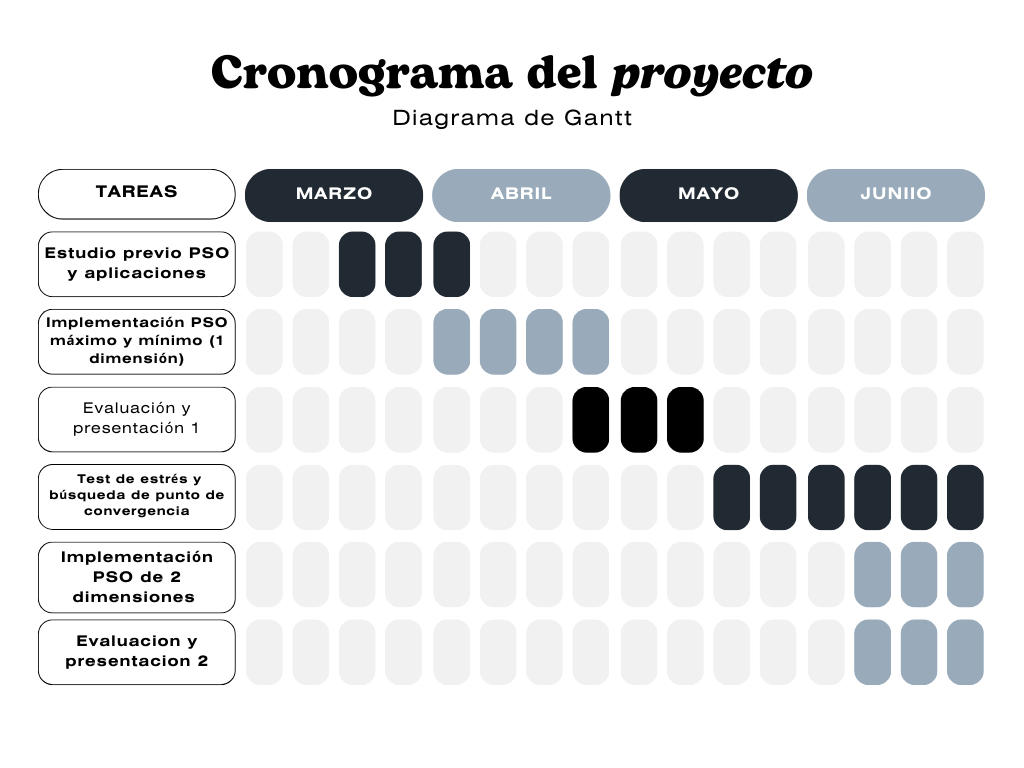
    print("Valor de función encontrado:", best\_fitness)

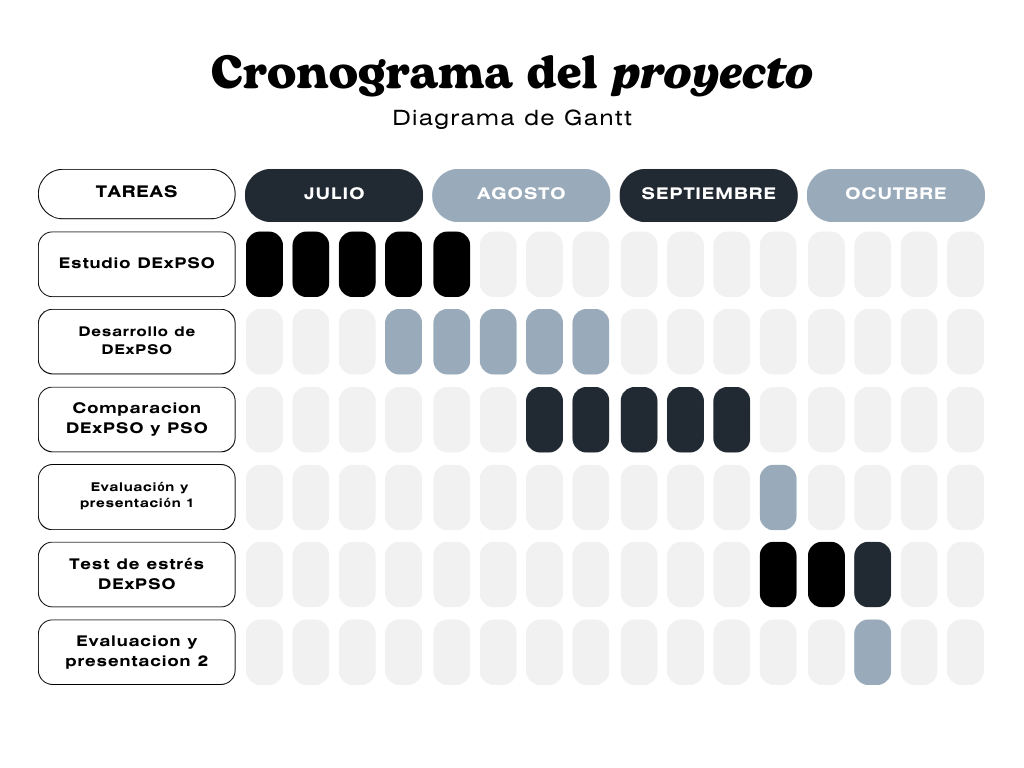
    if r > 3:

        break

print("Tiempo de ejecución:", execution\_time, "segundos")

# Cronograma del proyecto:







# Bibliografía

1. Some Properties of D-optimal Designs for Random Fields with Different Variograms (Milan Stehl´ık).
2. Caparrini, Fernando Sancho. (2019). PSO: Optimización por ejambres de partículas. En Fernando Sancho Caparrini. Recuperado el 1 de mayo de 2023, de <http://www.cs.us.es/~fsancho/?e=70>
3. Análisis de Convergencia Temprana en Algoritmos PSO con Función Objetiv Lineal (Miguel A. Azar, FabiolaP. Paz, Analía Herrera Cognetta).

<http://sedici.unlp.edu.ar/bitstream/handle/10915/45374/Documento_completo.pdf?sequence=1&isAllowed=y>

1. A Novel Double Exponential Particle Swarm Optimization (DExPSO) with Guaranteed Convergence and Applications to Find Optimal Exact Design(Milan Stehlík, Weng Kee Wong , Ping-Yang Chen and Jozef Kiseľák)
2. Particle Swarm Optimization with Various Inertia Weight Variants for Optimal Power Flow Solution (Prabha Umapathy, C. Venkataseshaiah, and M. Senthil Arumugam)