Universidad Nacional Autónoma de México

Facultad de Ciencias (2025-1)

Cómputo Evolutivo

MATERIAL DE EXPOSICIÓN: REPORTE

Algoritmos Bio Inspirados: Comparación PSO en optimización Continua

---

Flores Linares Oscar Daniel 320208591

27 de Noviembre del 2025

1. Introducción

En este reporte vamos a mostrar el desarrollo y los resultados basados en el tema de mi proyecto final. Aquí mostraremos de forma práctica los conceptos de un algoritmo Bio Inspirado llamado Algoritmo de Optimización por Cúmulo de Partículas (PSO) y compararemos este mismo contra un Algoritmo Genético que se adaptó a optimización continua.

Aquí no solo implementamos el algoritmo PSO estándar, sino que se programaron 3 variantes diferentes modificando sus parámetros. De esta manera vamos a poder ver como afectan el comportamiento de enjambre y que se vuelvan más exploradores o explotadores.

También se pusieron a prueba los algoritmos con 5 funciones de prueba clásicas con una dimensión de 10 y de 30, más que nada para ver la robustez del problema cuando se hace más complejo.

Explicaremos cada clase implementada y analizaremos las gráficas de convergencia y boxplots.

2. Desarrollo

2.1. Funciones de prueba

Para poder ver como funcionan nuestros algoritmos, implementamos una clase que va a tener 5 funciones benchmark.

- **Función Esfera (Sphere)**:
  Esta se representa como un paraboloide convexo y unimodal. Solo tiene un mínimo global en el origen. No tiene mínimos locales que confundan al algoritmo. En esta función probar la explotación es la mejor opción si queremos saber que tan capaz es, y viable para saber que no tiene errores básicos el algoritmo.

  En el código lo implementamos como una suma de los cuadrados de los elementos del vector.

- **Función Ackley (ackley)**:
  Esta función es más compleja, pues tiene la superficie plana en los exteriores con pequeños baches. También cuenta con un agujero en el centro. Es la combinación de una exponencial con un coseno. Cosa que es un reto para el algoritmo pues tiene el riesgo que quedarse en algunos de los baches de los exteriores.

  En el código dividimos en `termino1` que es para la parte exponencial y `termino2` que es la parte del coseno.

- **Función Griewank (griewank)**:
  Esta función tiene muchos mínimos locales, y la dificultad de la función se hace más pequeña mientras la dimensión se hace más grande. En dimensiones pequeñas se hace engañosa por el ruido del producto de cosenos.

  En el código tenemos `parte_suma` que lleva al centro y `parte_prod` que hace los mínimos locales. También tenemos `np.arange(1, len(x) + 1)` para los índices que son divisores en el producto y que la dimensión escale de diferente manera.

- **Función Rastrigin (rastrigin)**:
  En esta función es muy importante ver la capacidad de exploración pues es la más ideal. Tiene

forma de parábola pero hecha con una onda coseno, lo que hace un paisaje con muchos picos y valles. Si configuramos un algoritmo con poca exploración entonces va a caer en algún valle que tenga cerca.

En el código utilizamos una constante `A = 10` para poder controlar la amplitud de los picos. La linea más importante es `x**2 - A * np.cos(...)`, para cambiar de forma la superficie cuadrática.

- **Función Rosenbrock (rosenbrock)**:
  Aquí contamos con un mínimo global con un fondo plano, el cual está en un valle largo y parabólico. Es demasiado difícil converger en el mínimo global porque el gradiente casi es nulo, pero encontrar el valle no representa una dificultad.

  En el código usamos listas del estilo `x[1:]` y `x[:-1]` para poder hacer el cálculo entre la variable i y la i + 1, en la línea `100.0 * (x[1:] - x[:-1]**2)**2...`

2.1.1. Pseudocódigo

Algorithm 1 Biblioteca de Funciones Benchmark para Optimización Continua

Vector solución candidato x de dimensión D.

Definición de Funciones Matemáticas:

Function Esfera(x)
return $\sum_{i=1}^{D} x_i^2$
End Function
Function Ackley(x)
Definir constantes $a \leftarrow 20, b \leftarrow 0{,}2, c \leftarrow 2\pi$
$S_1 \leftarrow \sum_{i=1}^{D} x_i^2$
$S_2 \leftarrow \sum_{i=1}^{D} \cos(c \cdot x_i)$
return $-a \cdot \exp\left(-b\sqrt{\frac{S_1}{D}}\right) - \exp\left(\frac{S_2}{D}\right) + a + e$
End Function
Function Griewank(x)
$S \leftarrow \frac{1}{4000} \sum_{i=1}^{D} x_i^2$
$P \leftarrow \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right)$
return $S - P + 1$
End Function
Function Rastrigin(x)
Definir constante $A \leftarrow 10$
return $A \cdot D + \sum_{i=1}^{D} (x_i^2 - A \cdot \cos(2\pi x_i))$
End Function
Function Rosenbrock(x) ▷ Suma de términos adyacentes
return $\sum_{i=1}^{D-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$
End Function

Método Auxiliar:

Function ObtenerInfo(nombre)
Definir mapa de configuración M:
'Esfera': $f \leftarrow$ Esfera, limites $\leftarrow (-100, 100)$
'Ackley': $f \leftarrow$ Ackley, limites $\leftarrow (-32, 32)$
'Griewank': $f \leftarrow$ Griewank, limites $\leftarrow (-600, 600)$
'Rastrigin': $f \leftarrow$ Rastrigin, limites $\leftarrow (-5{,}12, 5{,}12)$
'Rosenbrock': $f \leftarrow$ Rosenbrock, limites $\leftarrow (-30, 30)$
return M[nombre]
End Function

2.2. Algoritmo PSO

Aquí implementamos la estrategia de enjambre donde definimos a los exploradores.

Primero creamos el enjambre con N partículas que van a estar en un espacio donde la dimensión la definimos nosotros para poder hacer la búsqueda.

Aquí modelé el enjambre con matrices en vez de hacer objetos para partículas individuales. Con `self.posiciones` hacemos una matriz (N X D) en donde cada fila es representada como partícula y las inicializamos de forma aleatoria para que abarque una buena cantidad al inicio.

Para las velocidades aplicamos una matriz del mismo tamaño, y se definió con valores pequeños entre -1 y 1, de manera que las partículas inicien con un impulso aleatorio.

Para la memoria del enjambre tenemos 2 parámetros:

- `pbest (mejor personal)`: Así toda partícula puede recordar donde estuvo el mejor hallazgo, en donde por cierto el inicio se define como su posición actual.
- `gbest (Mejor Global)`: Aquí es la mejor solución que se encontró por cualquiera del enjambre. Esto se obtiene buscando el valor más pequeño de fitness que inició.

Despues tenemos `ejecutar`, aquí aplicamos la ecuación más importante de PSO. Gracias a ello podemos actualizar la velocidad y la posición en cada iteración.

Para actualizar la velocidad, la cuál nos dice hacia donde y con que rapidez se moverá la partícula, con la suma de algunos componentes:

- `inercia (self.w * self.velocidades)`: Aquí se muestra la tendencia de la partícula en base a la dirección que llevaba. 'w' define el freno o derrape.
- `Componente cognitivo (c1 * r1 * (pbest - x))`: Aquí se muestra el comportamiento de la partícula en base al mejor recuerdo que tiene. Mismo que lo atrae hacia él con `self.mejor_personal_pos`. 'r1' funciona como el factor aleatorio variando el movimiento.
- `Componente Social (c2 * r2 * (gbest - x))`: Este componente atrae la partícula para el lidel del enjambre con `self.mejor`$_{g}lobal_{p}os$

Para actualizar la posición es algo más sencillo una vez se calcula la nueva velocidad:

$$x(t+1) = x(t) + v(t+1)$$

lo que se representa en el código con `self.posiciones = self.posiciones + self.velocidades`.

También controlamos las fronteras, pues las particulas se pueden salir del espacio de busqueda que definimos si va muy rápido. Por eso con nuestra función `np.clip`, si alguna partícula intenta salir del límite, entonces se quedará pegada en el borde hasta que haya una nueva velocidad que la vaya regresando al centro.

Una vez que se mueven las partículas, estas van a calcular su fitness, y van ocurriendo 2 actualizaciones importantes:

- **Actualización de pbest**: Se compara el fitness actual con el histórico con la línea `mejoras = fitness_actuales <self.mejor_personal_fit` y con ello actualizar las partículas que sí presentaron una mejoría.
- **Actualización de gbest:** Si la mejor partícula es mejor que el lider histórico, se actualizan las coordenadas y el óptimo global.

Al final regresamos la mejor solución, el tiempo de cómputo e historial de convergencia.

2.2.1. Pseudocódigo

Algorithm 2 Optimización por Cúmulo de Partículas (PSO)

Función objetivo f, dimensión D, límites $[L, U]$, tamaño enjambre N, iteraciones $T_{\text{máx}}$, coeficientes w, c_1, c_2. Mejor solución global g_{best} y su aptitud.

Inicializar posiciones $X \sim U(L, U)^{N \times D}$ y velocidades $V \sim U(-1, 1)^{N \times D}$
Inicializar mejores personales $P \leftarrow X$
Evaluar aptitudes iniciales y determinar mejor global g_{best}
for $t = 1$ **to** $T_{\text{máx}}$ **do**
    Generar matrices aleatorias $R_1, R_2 \sim U(0, 1)^{N \times D}$
    Actualizar Velocidades:
    $V \leftarrow wV + c_1 R_1 \odot (P - X) + c_2 R_2 \odot (g_{best} - X)$
    Actualizar Posiciones:
    $X \leftarrow X + V$
    Aplicar límites: $X \leftarrow \text{máx}(\text{mín}(X, U), L)$
    Evaluar y Actualizar:
    for cada partícula $i \in \{1, \ldots, N\}$ **do**
        $fitness \leftarrow f(x_i)$
        if $fitness < f(p_i)$ **then**
            $p_i \leftarrow x_i$ ▷ Actualizar mejor personal
            if $fitness < f(g_{best})$ **then**
                $g_{best} \leftarrow x_i$ ▷ Actualizar mejor global
            end if
        end if
    end for
end for
return $g_{best}, f(g_{best})$

2.3. Algoritmo genético con optimización continua

Para lograr la comparación entre un algoritmo PSO y uno genético, tuvimos que adaptar el genético pues en anteriores ocasiones lo hicimos sobre un problema combinatorio.

Primero para la representación de un individuo usamos un vector solución $x \in \mathbb{R}^D$. El **genotipo** se represento con un arreglo llamado `self.genotipo` para hacer operaciones matemáticas en el cromosoma sin ir gen por gen.
La **inicialización** nos va a dar valores aleatorios dentro de los límites definidos, y que todo se pueda realizar en el espacio de búsqueda.

Utilizamos **selección por torneo** el cual selecciona a los papás que van a padar los genes a sus hijos. Elegimos k individuos que competirán entre ellos. El que tenga mejor fitness gana para poder reproducirse.
En el código hice un torneo de 3 con `size=6`, esto para tomar 2 papás de golpe por grupos de 3. Así podemos ir moldeando la presión selectiva y no hay problemas para escalar, que tal vez sí tendría otro tipo de selección como el de ruleta para algunas funciones.

Tomamos un operador de cruza aritmética definida como `_cruza_aritmetica` obteniendo al hijo

como un promedio ponderado de padres P_1 y P_2:

$$H_1 = \alpha \cdot P_1 + (1 - \alpha) \cdot P_2$$

$$H_2 = (1 - \alpha) \cdot P_1 + \alpha \cdot P_2$$

Donde α puede ser cualquier número ente 0 y 1.
En el código hacemos vectorizada la operación con `alpha * p1.genotipo + (1 - alpha) * p2.genotipo` para obtener descendencia con explotación en las zonas que más prometen.

El operador de mutación fue gaussiano y representado en código en la línea `_mutacion_gaussiana` para poder tener diversidad y tratar de evitar los óptimos locales. Por lo que esa adición de ruido hacia el valor del gen fue Ruido Gaussiano $N(0, \sigma)$.
En el código lo que hago es decidir antes que nada los genes que mutarán por cada dimensión con `mask = rng.random <prob_mut`. A esos genes sumamos un valor aleatorio con distribución normal y definimos la fuerza de la mutación al 10 % del rango de busqueda. También usamos `np.clip` por cualquier gen fuera de los límites.

La línea `ejecutar` hace una nueva población en cada iteración, e implementamos **elitismo** para que la solución nunca empeore, por lo que antes de usar la selección y cruza se copia al mejor individuo actual con `copy.deepcopy(mejor_actual)` y este lo llevamos a la siguiente generación para obtener convergencia monótona.

2.3.1. Pseudocódigo

Algorithm 3 Algoritmo Genético con Codificación Real (RC-GA)

Función f, dimensión D, límites $[L, U]$, población N, generaciones $T_{\text{máx}}$, p_c, p_m. Mejor solución global g_{best}.

Inicializar población $P \leftarrow \{x_1, \ldots, x_N\}$ con $x_i \sim U(L, U)^D$
Evaluar P y encontrar el mejor individuo g_{best}
Definir desviación estándar de mutación $\sigma \leftarrow 0{,}1 \cdot (U - L)$
for $t = 1$ **to** $T_{\text{máx}}$ **do**
    $P' \leftarrow \emptyset$
    $x_{elite} \leftarrow \text{argmin}_{x \in P} f(x)$
    $P' \leftarrow P' \cup \{x_{elite}\}$ ▷ Elitismo: preservar al mejor
    while $|P'| < N$ **do**
        Selección por Torneo ($k = 3$):
        Seleccionar p_1, p_2 de P ganadores de torneos aleatorios
        Cruza Aritmética:
        if $rand(0, 1) < p_c$ **then**
            $\alpha \sim U(0, 1)$
            $h_1 \leftarrow \alpha p_1 + (1 - \alpha) p_2$
            $h_2 \leftarrow (1 - \alpha) p_1 + \alpha p_2$
        else
            $h_1, h_2 \leftarrow p_1, p_2$
        end if
        Mutación Gaussiana:
        for cada hijo $h \in \{h_1, h_2\}$ y cada gen $j \in \{1, \ldots, D\}$ **do**
            if $rand(0, 1) < p_m$ **then**
                $h[j] \leftarrow h[j] + N(0, \sigma)$ ▷ Ruido Gaussiano
                $h[j] \leftarrow \text{máx}(\text{mín}(h[j], U), L)$ ▷ Límites
            end if
        end for
        Evaluar $f(h_1), f(h_2)$
        $P' \leftarrow P' \cup \{h_1, h_2\}$
    end while
    $P \leftarrow P'$ ▷ Reemplazo generacional
    Actualizar g_{best} con el mejor de P
end for
return g_{best}

2.4. Validación preliminar de PSO

Antes de hacer el main final para ver el comparativo entre los diferentes parámetros de PSO y el algoritmo genético, lo que hice fue hacer un main para ver la integridad de la clase PSO y verificar que el enjambre pueda respetar los límites de cada función y reducir el fitness. Aquí manejamos 3 etapas:

- Definimos hiperparámetros globales de dimensión 10 y 30 partículas para una ejecución rápida.

- Nuestra clase se itera por cada una de las funciones utilizadas en el proyecto, las cuales están enlistadas.

- Usamos `FuncionesPrueba.obtener_info()` para obtener la función objetivo y sus límites, poniendolos en el constructor de PSO.

El resultado es un diagnóstico sencillo en donde si conseguimos un fitness menor a (10^{-5}) la consideramos como **Convergencia Exitosa**, de otra manera la consideramos parcial.

2.4.1. Pseudocódigo

Algorithm 4 Rutina de Validación Unitaria del PSO

Lista de funciones $F_{nombres}$, dimensión D, partículas N, iteraciones T.

Imprimir encabezado del experimento
for cada *nombre* en $F_{nombres}$ **do**
    Obtener Metadatos:
    $info \leftarrow$ FuncionesPrueba.obtener_info($nombre$)
    $f \leftarrow info.func$
    $[L, U] \leftarrow info.limites$
    Instanciación:
    $optimizador \leftarrow$ Nuevo PSO$(f, D, [L, U], N, T)$
    Ejecución:
    $g_{best}, fitness, historial \leftarrow optimizador.ejecutar()$
    Diagnóstico:
    Imprimir $fitness$ encontrado
    if $fitness < 10^{-5}$ **then**
        Imprimir "¡Convergencia Exitosa!"
    else
        Imprimir Çonvergencia Parcial"
    end if
end for
Terminar proceso

2.5. Experimentación y Ejecución (Main Principal)

:

Desarrollamos `main_comparativo.py` para obtener datos y visualizaciones para el análisis. Esto lo estrcuturamos de la siguiente forma:

- Primero evaluamos que tan robustos son los algoritmos. Definimos la dimensión, la cual para todas las funciones fueron en 10D y 30D, lo cual hace que el espacio de busqueda crezca exponencialmente y con eso podemos ver que algoritmos hacen buena escala y cuales no. También todos los experimentos se hicieron con 50 individuos y en 100 iteraciones para hacer una comparación justa y por último cada configuración se hace 30 veces para tener estadísticas más confiables.

- **Variantes de PSO**: Se definio un diccionario llamado `configuraciones_pso` con diferentes hiperparámetros a la clase PSO con **PSO estandar:**Balanceado ($w = 0{,}7$), **PSO Exploración:**

Inercia alta ($w = 0{,}9$) y mayor componente social ($c_2 > c_1$) y que sirve para recorrer distancias grandes sin óptimos locales, y **PSO Explotación:** Inercia baja ($w = 0{,}4$) y mayor componente cognitivo ($c_1 > c_2$), para soluciones en áreas que prometen.

- **Semillas**: Siendo una de las cosas más importantes, usamos una semilla base `(semilla_base = 12345 + rep)`, así aseguramos obtener los mismos resultados sin importar en que momento se vuelva a ejecutar, y usamos un offset `+ 10000` con el algoritmo genético para evitar que use la misma secuencia aleatoria de PSO.

- **Gráficas con datos**: Usamos `matplotlib.pyplot.boxplot` para generar gráficas que muestra como se dispersan las 30 ejecuciones, sabiendo que para la robustez, una caja pequeña significa que el algoritmo es confiable y persistente.
  También graficamos el promedio del mejor fitness durante las iteraciones con una escala de tipo logaritmica con `plt.yscale('log')`.

- **Ranking Global**: Al final implementamos un ranking que ordena tanto las variantes de PSO como el algoritmo genético según el desempeño promedio que tuvieron. Esto para saber que variante o algoritmo se comportó mejor en cada dimensión más allá de los resultados individuales por cada función.

2.5.1. Pseudocódigo

Algorithm 5 Diseño Experimental Comparativo (PSO vs AG)

Conjunto de dimensiones $\mathcal{D} = \{10, 30\}$, iteraciones $T_{\text{máx}} = 100$, población $N = 50$, repeticiones $K = 30$. Conjunto de funciones $\mathcal{F} = \{\text{Esfera, Ackley, Griewank, Rastrigin, Rosenbrock}\}$. Conjunto de algoritmos $\mathcal{A} = \{PSO_{std}, PSO_{exp}, PSO_{explot}, AG_{real}\}$.

Inicializar contenedores de resultados globales
for cada dimensión $D \in \mathcal{D}$ **do**
  for cada función $f \in \mathcal{F}$ **do**
    Obtener límites $[L, U]$ para f
    Inicializar matriz de resultados $M_{f,D}$
    for cada algoritmo $A \in \mathcal{A}$ **do**
      for $k = 1$ **to** K **do**
        Definir semilla aleatoria $S_k \leftarrow \text{Base} + k$ ▷ Garantizar reproducibilidad
        if A es AG **then**
          $S_k \leftarrow S_k + \text{Offset}$ ▷ Independencia estadística
        end if
        Ejecutar:
        $best, fitness, tiempo \leftarrow A.\text{ejecutar}(f, D, [L, U], N, T_{\text{máx}}, S_k)$
        Almacenar $(fitness, tiempo)$ en $M_{f,D}[A]$
      end for
      Calcular Media μ_A y Desviación Estándar σ_A de los fitness
    end for
    Análisis Estadístico y Visualización:
    Generar Diagrama de Caja (Boxplot) con $M_{f,D}$ ▷ Evaluar Robustez
    Graficar Curvas de Convergencia Promedio (log scale) ▷ Evaluar Velocidad
    Calcular Ranking de algoritmos basado en μ_A (menor es mejor)
  end for
  Consolidar Rankings promedio para la dimensión D
end for
Reportar Tablas de Rankings, Tiempos medios y Gráficas generadas

3. Tabla con resultados

Cuadro 1: Resultados comparativos de algoritmos de optimización

Dimensión	Función	Algoritmo	Variante	Media	Desv. Est.	Tiempo (s)
10	Esfera	PSO	Estándar	1.723e-03	1.582e-03	0.024
			Exploración	2.244e+03	6.453e+02	0.024
			Explotación	3.299e+02	2.886e+02	0.024
		AG	Base	7.746e-01	4.285e-01	0.609
	Ackley	PSO	Estándar	1.685e-02	1.021e-02	0.081
			Exploración	1.468e+01	1.823e+00	0.080
			Explotación	5.269e+00	1.862e+00	0.079
		AG	Base	6.478e-01	3.060e-01	0.693
	Griewank	PSO	Estándar	2.364e-01	1.279e-01	0.059
			Exploración	2.534e+01	1.520e+01	0.058
			Explotación	3.402e+00	2.887e+00	0.058
		AG	Base	7.109e-01	1.852e-01	0.659
30	Esfera	PSO	Estándar	1.208e+02	1.875e+03	0.025
			Exploración	5.307e+04	6.004e+03	0.025
			Explotación	6.843e+03	1.685e+03	0.025
		AG	Base	1.909e+02	4.503e+01	0.631
	Ackley	PSO	Estándar	5.301e+00	3.168e+00	0.081
			Exploración	2.009e+01	2.102e-01	0.080
			Explotación	1.266e+01	1.037e+00	0.080
		AG	Base	4.684e+00	3.622e-01	0.708
	Griewank	PSO	Estándar	3.777e+00	1.690e+01	0.062
			Exploración	4.819e+02	6.469e+01	0.062
			Explotación	6.150e+01	1.350e+01	0.061
		AG	Base	2.657e+00	3.872e-01	0.688

Nota: Los valores de Media y Desv. Est. corresponden al mejor valor final obtenido en cada ejecución. El tiempo se reporta en segundos como promedio de 30 repeticiones.

Cuadro 2: Resultados para funciones Rastrigin y Rosenbrock

Dimensión	Función	Algoritmo	Variante	Media	Desv. Est.	Tiempo (s)
10	Rastrigin	PSO	Estándar	1.183e+01	5.450e+00	0.046
			Exploración	8.194e+01	1.454e+01	0.045
			Explotación	1.096e+01	4.215e+00	0.047
		AG	Base	5.278e+00	2.136e+00	0.637
	Rosenbrock	PSO	Estándar	4.329e+02	1.020e+03	0.060
			Exploración	8.267e+05	6.453e+05	0.054
			Explotación	2.550e+04	3.059e+04	0.054
		AG	Base	4.174e+01	5.220e+01	0.645
30	Rastrigin	PSO	Estándar	1.494e+02	2.687e+01	0.049
			Exploración	3.943e+02	3.239e+01	0.048
			Explotación	9.267e+01	2.582e+01	0.049
		AG	Base	1.001e+02	1.554e+01	0.677
	Rosenbrock	PSO	Estándar	2.586e+04	4.239e+04	0.058
			Exploración	1.483e+08	5.075e+07	0.057
			Explotación	3.439e+06	1.877e+06	0.057
		AG	Base	4.856e+03	2.225e+03	0.685

Referencias

[1] Eiben, A. E., & Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer.

[2] Engelbrecht, A. P. (2005). *Fundamentals of Computational Swarm Intelligence*. Wiley.

[3] Kennedy, J., & Eberhart, R. (1995). Particle Swarm Optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*.