



## Instituto Politecnico Nacional Escuela Superior de Computo

Materia: Tópicos Selectos de Algoritmos Bioinspirados  
Grupo: 7BM1

Profesor: Daniel Molina Pérez  
Periodo: 2025/02

### Practica 02

#### *Optimización de Carteras: Un Enfoque de Algoritmo Genético con Manejo de Restricciones para la Selección de Activos*

Realizado por:  
Carrillo Barreiro José Emiliano  
Martinez Ayala Gerardo  
Robles Otero José Ángel  
Vásquez Morales Haniel Ulises

#### **Abstract:**

This paper presents an approach to solve the portfolio optimization problem using a genetic algorithm with constraint handling. The goal of portfolio optimization is to find the optimal allocation of capital among different financial assets to maximize return and/or minimize risk. The genetic algorithm is implemented to address this problem, incorporating constraint handling through the exterior penalty method to ensure that solutions satisfy real-world limitations, such as investment limits per asset and acceptable risk levels. The results obtained from applying the algorithm to a stock selection case study are presented and discussed.

#### **Resumen:**

Esta práctica aborda el problema de la optimización de carteras de inversión, donde el objetivo es determinar la distribución óptima del capital entre diferentes activos financieros para maximizar el retorno y/o minimizar el riesgo. Se implementa un algoritmo genético para resolver este problema, incorporando el manejo de restricciones mediante el método de penalización exterior para asegurar que las soluciones obtenidas cumplan con las limitaciones del mundo real, como límites de inversión por activo y niveles de riesgo aceptables. Se presentan y discuten los resultados obtenidos al aplicar el algoritmo a un caso de estudio de selección de acciones.

**Fecha: 14 de abril de 2025**

# Índice general

<b>1</b>	<b>Introducción</b>	<b>1-1</b>
<b>2</b>	<b>Objetivos del Proyecto</b>	<b>2-1</b>
2.1	Objetivo general . . . . .	2-1
2.2	Objetivo del inciso uno . . . . .	2-1
2.3	Objetivo inciso dos . . . . .	2-1
2.4	Objetivo inciso tres . . . . .	2-1
<b>3</b>	<b>Metodología</b>	<b>3-1</b>
3.1	Inicialización de la Población . . . . .	3-1
3.2	Evaluación de Fitness . . . . .	3-1
3.3	Selección por Torneo . . . . .	3-3
3.4	Cruzamiento con SBX . . . . .	3-3
3.5	Mutación Polinomial . . . . .	3-4
3.6	Elitismo y Ciclo Evolutivo . . . . .	3-4
3.7	Penalización Exterior . . . . .	3-5
<b>4</b>	<b>Resultados y Discusión</b>	<b>4-1</b>
4.1	Análisis de los Resultados . . . . .	4-2
4.1.1	Consistencia y Robustez . . . . .	4-2
4.1.2	Análisis por Problema . . . . .	4-2
4.1.3	Evolución del Fitness y Visualizaciones . . . . .	4-2
4.2	Discusión Resultados . . . . .	4-4
<b>5</b>	<b>Implementación</b>	<b>5-1</b>
5.1	Funciones Objetivo . . . . .	5-1
5.1.1	Descripción . . . . .	5-1
5.1.2	Implementación . . . . .	5-1
5.2	Módulos del Algoritmo Genético . . . . .	5-1
5.2.1	Inicialización . . . . .	5-1
5.2.2	Selección . . . . .	5-2
5.2.3	Cruzamiento . . . . .	5-2
5.2.4	Mutación . . . . .	5-2
5.2.5	Aplicación de Restricciones . . . . .	5-2
5.2.6	Ejecución del Algoritmo . . . . .	5-2
5.3	Visualización y Almacenamiento . . . . .	5-2
5.3.1	Visualización . . . . .	5-2
5.3.2	Almacenamiento . . . . .	5-2
<b>6</b>	<b>Conclusiones</b>	<b>1</b>
<b>A</b>	<b>GitHub</b>	<b>A-1</b>
<b>B</b>	<b>Scripts</b>	<b>B-1</b>
B.1	Archivo <code>main.py</code> . . . . .	B-1
B.2	Archivo <code>AG_confs.py</code> . . . . .	B-3
B.3	Archivo <code>AG.py</code> . . . . .	B-3
B.4	Archivo <code>selection.py</code> . . . . .	B-5



B.5	Archivo <code>crossover.py</code> . . . . .	B-6
B.6	Archivo <code>mutation.py</code> . . . . .	B-8
B.7	Archivo <code>auxiliares_functions.py</code> . . . . .	B-9
<b>C</b>	<b>Registro de indicadores completo.</b>	<b>C-1</b>
C.1	Problema 01 . . . . .	C-1
C.1.1	Resúmenes . . . . .	C-1
C.2	Problema 02 . . . . .	C-2
C.2.1	Resúmenes . . . . .	C-2
C.3	Problema 03 . . . . .	C-3
C.3.1	Resúmenes . . . . .	C-3

# Índice de cuadros

4.1	Problema 01: resumen_global.corridas	4-1
4.2	Problema 02: resumen_global.corridas	4-1
4.3	Problema 03: resumen_global.corridas	4-1
C.1	Archivo: resumen_run_1	C-1
C.2	Archivo: resumen_run_2	C-1
C.3	Archivo: resumen_run_3	C-1
C.4	Archivo: resumen_run_4	C-1
C.5	Archivo: resumen_run_5	C-2
C.6	Archivo: resumen_run_1	C-2
C.7	Archivo: resumen_run_2	C-2
C.8	Archivo: resumen_run_3	C-2
C.9	Archivo: resumen_run_4	C-2
C.10	Archivo: resumen_run_5	C-2
C.11	Archivo: resumen_run_1	C-3
C.12	Archivo: resumen_run_2	C-3
C.13	Archivo: resumen_run_3	C-3
C.14	Archivo: resumen_run_4	C-3
C.15	Archivo: resumen_run_5	C-3

# Índice de figuras

4.1	Evolución del fitness del Problema 1	4-3
4.2	Evolución del fitness del Problema 2	4-3
4.3	Evolución del fitness del Problema 3	4-4

## Capítulo 1

# Introducción

En el presente trabajo se implementa y analiza un algoritmo genético (AG) como herramienta de optimización para resolver un problema restringido proveniente del ámbito financiero. Los algoritmos genéticos, inspirados en los principios de la evolución biológica, constituyen una técnica robusta de búsqueda estocástica ampliamente utilizada en la optimización de funciones no lineales, especialmente en contextos donde las restricciones y la dimensionalidad del espacio de búsqueda dificultan la aplicación de métodos deterministas tradicionales. La arquitectura del algoritmo propuesto incorpora operadores clásicos de selección por torneo, cruzamiento mediante Simulated Binary Crossover (SBX) con preservación de límites, y mutación polinomial, todos ellos diseñados para mantener la diversidad poblacional y facilitar la exploración eficiente del espacio de soluciones. Además, se introduce un enfoque de penalización basado en la agregación de violaciones de restricciones, ponderadas por un parámetro  $\lambda$ , que permite adaptar el esquema de evaluación de aptitud a entornos restringidos. El sistema ha sido diseñado para ejecutarse múltiples veces sobre una o más funciones objetivo, con el fin de obtener estadísticas robustas sobre el desempeño del AG. Para cada corrida, se registra el historial evolutivo del mejor individuo por generación, el comportamiento de las violaciones de restricciones y los valores estadísticos agregados (mejor, peor, promedio y desviación estándar). Asimismo, se generan representaciones gráficas que permiten visualizar la evolución del fitness y la viabilidad de las soluciones, así como superficies tridimensionales del espacio de búsqueda para funciones con dos variables.

## Capítulo 2

# Objetivos del Proyecto

### 2.1 Objetivo general

Desarrollar e implementar un algoritmo genético para la resolución de un problema de optimización restringida en el ámbito financiero, con el fin de determinar la cartera de inversión óptima bajo diferentes criterios de maximización de retorno y control de riesgo.

### 2.2 Objetivo del inciso uno

Determinar la asignación óptima de capital en una cartera compuesta por seis acciones, maximizando el retorno esperado bajo la restricción de no invertir más del 40

### 2.3 Objetivo inciso dos

Diseñar una estrategia de inversión que minimice el riesgo asociado a la cartera, garantizando un rendimiento esperado mínimo del 35

### 2.4 Objetivo inciso tres

Identificar la combinación de inversión que maximice el rendimiento esperado de la cartera sin sobrepasar un nivel de riesgo del 20

## Capítulo 3

# Metodología

### 3.1 Inicialización de la Población

La población inicial se genera de forma uniforme a lo largo del espacio de búsqueda, definido por límites inferiores y superiores para cada variable.

#### Objetivo

Garantizar que la búsqueda comience explorando de manera equitativa todas las regiones posibles, evitando sesgos que puedan limitar la diversidad de soluciones iniciales.

#### Implementación

Se utiliza la función `initialize_population`, la cual emplea métodos de generación aleatoria (por ejemplo, la función `np.random.uniform` de NumPy) para crear un conjunto de individuos.

#### Ventajas

- Permite cubrir todo el rango definido para cada variable.
- Aumenta la probabilidad de encontrar regiones prometedoras del espacio de soluciones desde el inicio.

### 3.2 Evaluación de Fitness

Cada individuo generado se evalúa mediante una función objetivo, la cual cuantifica la calidad de la solución propuesta. A continuación, se describen las funciones objetivo y las restricciones empleadas para cada problema abordado:

#### Funciones Utilizadas por Problema

- Problema 1: Optimización de Ganancias

- **Función Objetivo ( $f_{\text{finanzas}}(x)$ ):** El objetivo es maximizar las ganancias, calculado como una combinación lineal de las variables de decisión ( $x$ ), ponderadas por los siguientes coeficientes: 0.2, 0.42, 1, 0.5, 0.46 y 0.3 para  $x_0$  a  $x_5$  respectivamente. Matemáticamente, se expresa como:

$$f_{\text{finanzas}}(x) = -(0.2x_0 + 0.42x_1 + 1x_2 + 0.5x_3 + 0.46x_4 + 0.3x_5)$$

El signo negativo indica que, aunque la función se define para minimizar, en este contexto buscamos maximizar las ganancias, por lo que el algoritmo de optimización buscará minimizar el negativo de esta función.

- **Restricción de Igualdad ( $h_1(x)$ ):** Se impone una restricción de igualdad que requiere que la suma de todas las variables de decisión sea igual a 1:

$$h_1(x) = 1 - \sum_{i=0}^5 x_i = 0$$

- **Función Secundaria Monitoreada ( $f_{\text{riesgos}}(x)$ ):** Adicionalmente, se monitorea la función de riesgos, aunque no se utiliza directamente en la evaluación de fitness para este problema. Esta función se calcula como:

$$f_{\text{riesgos}}(x) = x^T \Sigma x$$

donde  $\Sigma$  representa la matriz de covarianza de los activos.

#### ■ Problema 2: Minimización de Riesgos

- **Función Objetivo ( $f_{\text{riesgos}}(x)$ ):** El objetivo principal es minimizar el riesgo, calculado mediante la siguiente función cuadrática:

$$f_{\text{riesgos}}(x) = x^T \Sigma x$$

donde  $\Sigma$  es la matriz de covarianza de los activos y  $x$  es el vector de las variables de decisión.

- **Restricción de Desigualdad ( $g_1(x)$ ):** Se establece una restricción de desigualdad que limita la ganancia mínima aceptable:

$$g_1(x) = 0.35 + f_{\text{finanzas}}(x) \leq 0$$

donde

$$f_{\text{finanzas}}(x) = -(0.2x_0 + 0.42x_1 + 1x_2 + 0.5x_3 + 0.46x_4 + 0.3x_5)$$

- **Restricción de Igualdad ( $h_1(x)$ ):** Al igual que en el Problema 1, se aplica la restricción de igualdad:

$$h_1(x) = 1 - \sum_{i=0}^5 x_i = 0$$

- **Función Secundaria Monitoreada ( $f_{\text{finanzas}}(x)$ ):** Se realiza un seguimiento de la función de ganancias para este problema, aunque no influye directamente en la optimización del riesgo.

#### ■ Problema 3: Optimización de Ganancias con Restricción de Riesgo

- **Función Objetivo ( $f_{\text{finanzas}}(x)$ ):** Similar al Problema 1, el objetivo es maximizar las ganancias:

$$f_{\text{finanzas}}(x) = -(0.2x_0 + 0.42x_1 + 1x_2 + 0.5x_3 + 0.46x_4 + 0.3x_5)$$

- **Restricción de Desigualdad ( $g_2(x)$ ):** Se introduce una restricción de desigualdad que establece un límite superior para el riesgo:

$$g_2(x) = 0.2 - x^T \Sigma x \leq 0$$



- **Restricción de Igualdad** ( $h_1(x)$ ): Se mantiene la restricción de que la suma de las variables de decisión debe ser igual a 1:

$$h_1(x) = 1 - \sum_{i=0}^5 x_i = 0$$

- **Función Secundaria Monitoreada** ( $f_{\text{riesgos}}(x)$ ): Se observa la función de riesgos como una métrica secundaria para este problema.

## Proceso

Se calcula el valor de fitness para cada individuo (por ejemplo, evaluando  $f(x_1, x_2)$ ) y se almacena dicho valor para posteriores comparaciones.

## Importancia

La evaluación correcta del fitness es crucial, ya que determina la selección de individuos y, por ende, el rumbo de la evolución poblacional.

# 3.3 Selección por Torneo

Para elegir los padres que generarán la siguiente generación se utiliza un método de selección por torneo.

## Mecanismo

- Se forman múltiples grupos (torneos) de individuos seleccionados al azar.
- En cada grupo se compara el fitness de los participantes y se selecciona al individuo con el mejor desempeño.

## Implementación Vectorizada

La función `vectorized_tournament_selection` realiza este proceso de forma eficiente, aprovechando operaciones vectorizadas de NumPy.

## Beneficios

- Favorece la selección de soluciones de alta calidad sin descartar por completo la diversidad poblacional.
- Permite controlar la presión selectiva mediante el tamaño del torneo.

# 3.4 Cruzamiento con SBX

El operador de cruzamiento se implementa mediante el método SBX (Simulated Binary Crossover).

## Proceso del SBX

- A partir de dos padres, se genera un número aleatorio  $u$  y se calcula un parámetro  $\beta$  que determina la dispersión de los descendientes respecto a los padres.
- Se generan dos hijos combinando linealmente los valores de los padres.

## Ajuste de Límites

Se incorpora un mecanismo en `sbx_crossover_with_boundaries` que garantiza que los hijos resultantes se mantengan dentro de los límites predefinidos.

## Ventajas

- Promueve la creación de soluciones intermedias que pueden explotar la información genética de ambos padres.
- Ayuda a preservar la diversidad en la población.

# 3.5 Mutación Polinomial

Para introducir variabilidad y explorar nuevas regiones del espacio de búsqueda, se aplica la mutación polinomial.

## Mecanismo de la Mutación

- Cada gen de un individuo tiene una probabilidad definida de sufrir una mutación.
- Se usa una distribución polinomial, controlada por el parámetro  $\eta_{mut}$ .

## Consideraciones de Límites

La mutación se aplica respetando los límites definidos para cada variable mediante la función `polynomial_mutation_with.L`

## Beneficios

- Introduce pequeñas variaciones que pueden conducir a la exploración de nuevas soluciones.
- Previene la convergencia prematura al mantener la diversidad genética.

# 3.6 Elitismo y Ciclo Evolutivo

El proceso evolutivo se estructura en ciclos o generaciones.

## Elitismo

- Se retiene el mejor individuo de la generación actual y se garantiza su inclusión en la siguiente generación.
- Esto asegura que la calidad de la solución nunca empeore a lo largo de las generaciones.

## Ciclo Evolutivo

- Cada generación incluye la selección, el cruzamiento, la mutación y la incorporación del individuo de élite.
- La evolución se repite durante un número predefinido de generaciones.

## Registro y Análisis

- Se almacena el historial del fitness y de las mejores soluciones.
- Esto facilita el análisis del comportamiento del algoritmo y la generación de visualizaciones.

## 3.7 Penalización Exterior

Los métodos de penalización son técnicas para manejar restricciones en problemas de optimización. La idea principal es transformar un problema de optimización con restricciones en uno sin restricciones, modificando la función objetivo para penalizar las soluciones que no cumplen con las restricciones. En el método de penalización exterior, esto se logra agregando términos a la función objetivo que aumentan su valor cuando se violan las restricciones.

La función objetivo modificada, denotada como  $F_P(x)$ , se define como:

$$F_P(x) = f(x) + \lambda P(x)$$

Donde:

- $f(x)$  es la función objetivo original del problema.
- $P(x)$  es la función de penalización, que cuantifica la violación de las restricciones.
- $\lambda_P$  es el parámetro de penalización, que controla la severidad de la penalización.

La función de penalización  $P(x)$  se calcula de la siguiente manera:

$$P(x) = \sum_{i=1}^n \max(0, g_i(x))^2 + \sum_{j=1}^m h_j(x)^2$$

Donde:

- $g_i(x)$  representa la  $i$ -ésima restricción de desigualdad, de la forma  $g_i(x) \leq 0$ .
- $h_j(x)$  representa la  $j$ -ésima restricción de igualdad, de la forma  $h_j(x) = 0$ .

## Funcionamiento de la Penalización Exterior

- Para las restricciones de desigualdad  $g_i(x) \leq 0$ :
  - Si la restricción se cumple,  $\max(0, g_i(x)) = 0$ , por lo que no hay penalización.
  - Si la restricción se viola,  $\max(0, g_i(x)) = g_i(x)$ , y se aplica una penalización proporcional al cuadrado de la violación.
- Para las restricciones de igualdad  $h_j(x) = 0$ :
  - Si la restricción se cumple,  $h_j(x)^2 = 0$ , y no hay penalización.
  - Si la restricción no se cumple,  $h_j(x)^2 > 0$ , y se aplica una penalización proporcional al cuadrado de la diferencia.

## Ventajas

- **Facilidad de implementación:** Es relativamente sencillo agregar la penalización externa a los algoritmos de optimización existentes.
- **Flexibilidad:** Permite explorar soluciones que violan las restricciones, lo cual puede ser útil en problemas complejos.

## Desventajas

- **Distorsión de la solución óptima:** La penalización puede llevar a soluciones subóptimas si se priorizan soluciones que violan las restricciones pero tienen un mejor valor de la función objetivo.
- **Sensibilidad al parámetro de penalización:** La elección del valor de  $\lambda_P$  es crucial. Si es muy pequeño, las restricciones pueden no cumplirse; si es muy grande, puede dificultar la convergencia.
- **Convergencia:** Puede causar una convergencia lenta o incluso la no convergencia del algoritmo.

## Capítulo 4

# Resultados y Discusión

Durante la ejecución del algoritmo genético, se realizaron múltiples corridas completas para cada uno de los tres problemas de optimización de carteras, lo que permitió evaluar la estabilidad y eficiencia del método. Los resultados se agruparon en resúmenes globales, donde se registraron indicadores clave, los cuales se encuentran dentro de las siguientes tablas:

Cuadro 4.1: Problema 01: resumen\_global.corridas

Indicador	x0	x1	x2	x3	x4	x5	Fitness	Fitness Secundario
Mejor (Fitness)	0.0	0.0	0.4	0.4	0.2024	0.0	-0.6925	0.0456
Peor (Fitness)	0.0	0.0968	0.4	0.1406	0.3603	0.0051	-0.6774	0.0533
Media	0.0	0.0754	0.4	0.325	0.1973	0.0052	-0.6854	0.0457
Desv. Estándar	0.0	0.0405	0.0	0.0958	0.085	0.0082	0.0053	0.0039

Cuadro 4.2: Problema 02: resumen\_global.corridas

Indicador	x0	x1	x2	x3	x4	x5	Fitness	Fitness Secundario
Mejor (Fitness)	0.2863	0.1552	0.0218	0.239	0.1511	0.1463	0.0021	-0.3772
Peor (Fitness)	0.2556	0.1245	0.0518	0.234	0.1825	0.1515	0.0031	-0.4016
Media	0.2417	0.1569	0.0312	0.2063	0.2092	0.1546	0.0026	-0.3912
Desv. Estándar	0.0357	0.0184	0.0107	0.0273	0.0403	0.0122	0.0004	0.0096

Cuadro 4.3: Problema 03: resumen\_global.corridas

Indicador	x0	x1	x2	x3	x4	x5	Fitness	Fitness Secundario
Mejor (Fitness)	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Peor (Fitness)	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Media	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Desv. Estándar	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Donde cada Indicador Representa lo siguiente:

- **Mejor (Fitness):** Representa la solución con el valor de fitness mínimo obtenido en todas las corridas.
- **Peor (Fitness):** Indica la solución con el mayor valor de fitness, sirviendo como referencia de la variabilidad en la búsqueda.
- **Media:** Es el promedio de los valores de fitness de la mejor solución de cada corrida, ofreciendo una visión global del desempeño del algoritmo.

- **Desv. Estándar:** Mide la dispersión de los valores de fitness entre las corridas, reflejando la estabilidad y consistencia del proceso evolutivo.

## 4.1 Análisis de los Resultados

### 4.1.1. Consistencia y Robustez

Los resúmenes globales muestran que, a lo largo de las corridas, el algoritmo tiende a converger de manera consistente hacia soluciones de alta calidad. Una baja desviación estándar en los valores de fitness sugiere que el proceso evolutivo es robusto y no depende en exceso de la aleatoriedad inherente a los operadores genéticos. Esto es crucial para problemas de optimización de carteras, ya que garantiza que la metodología aplicada es reproducible y confiable.

### 4.1.2. Análisis por Problema

#### Problema 1: Maximización de Retorno

El resumen global para el Problema 1 indica la capacidad del algoritmo para identificar carteras con un alto retorno esperado. El valor de “Mejor (Retorno)” obtenido se sitúa en un rango competitivo, y la media de las corridas respalda la eficacia del algoritmo genético para explorar el espacio de soluciones y encontrar combinaciones de activos que maximizan el retorno, respetando la restricción de inversión máxima por activo.

#### Problema 2: Minimización de Riesgo

Para el Problema 2, los resultados globales evidencian una convergencia hacia carteras con un riesgo mínimo, cumpliendo con la restricción de rendimiento esperado mínimo. Los indicadores de “Peor (Riesgo)” y “Desv. Estándar” muestran que, aunque existen ciertas variaciones entre corridas, el algoritmo genético logra encontrar soluciones consistentes que equilibran el riesgo y el rendimiento esperado.

#### Problema 3: Maximización de Retorno con Riesgo Controlado

En el Problema 3, los resultados muestran la capacidad del algoritmo para maximizar el retorno esperado sin exceder un nivel de riesgo predefinido. Esto demuestra la efectividad del método de penalización para guiar la búsqueda hacia soluciones que satisfacen simultáneamente múltiples objetivos y restricciones.

### 4.1.3. Evolución del Fitness y Visualizaciones

Las gráficas de la evolución del fitness, tanto en su forma original como normalizada, permiten observar el progreso generacional. Se aprecia una clara tendencia a la mejora, donde la mayoría de las corridas muestran una reducción significativa del valor de fitness a medida que avanzan las generaciones.

A continuación se dejan las gráficas de la evolución del fitness:

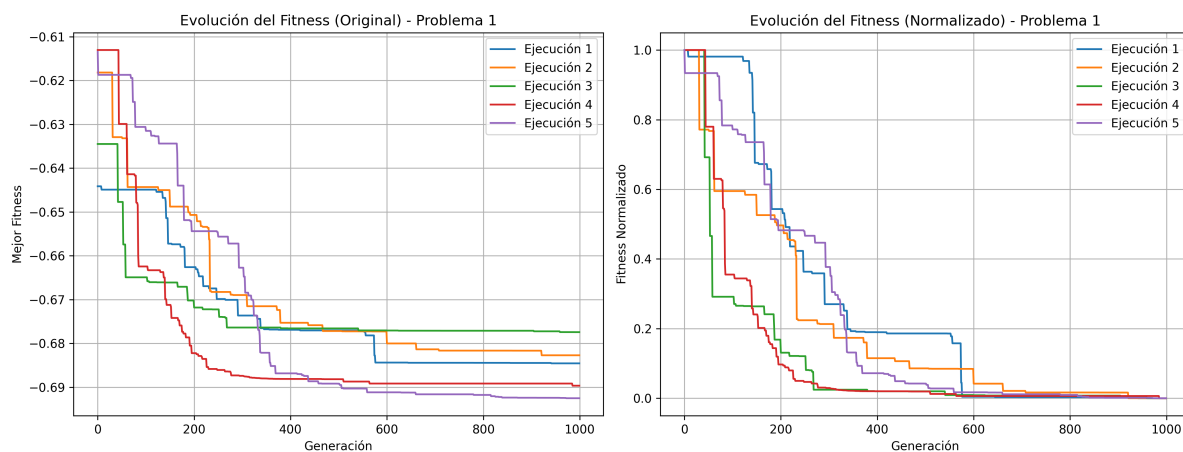


Figura 4.1: Evolución del fitness del Problema 1

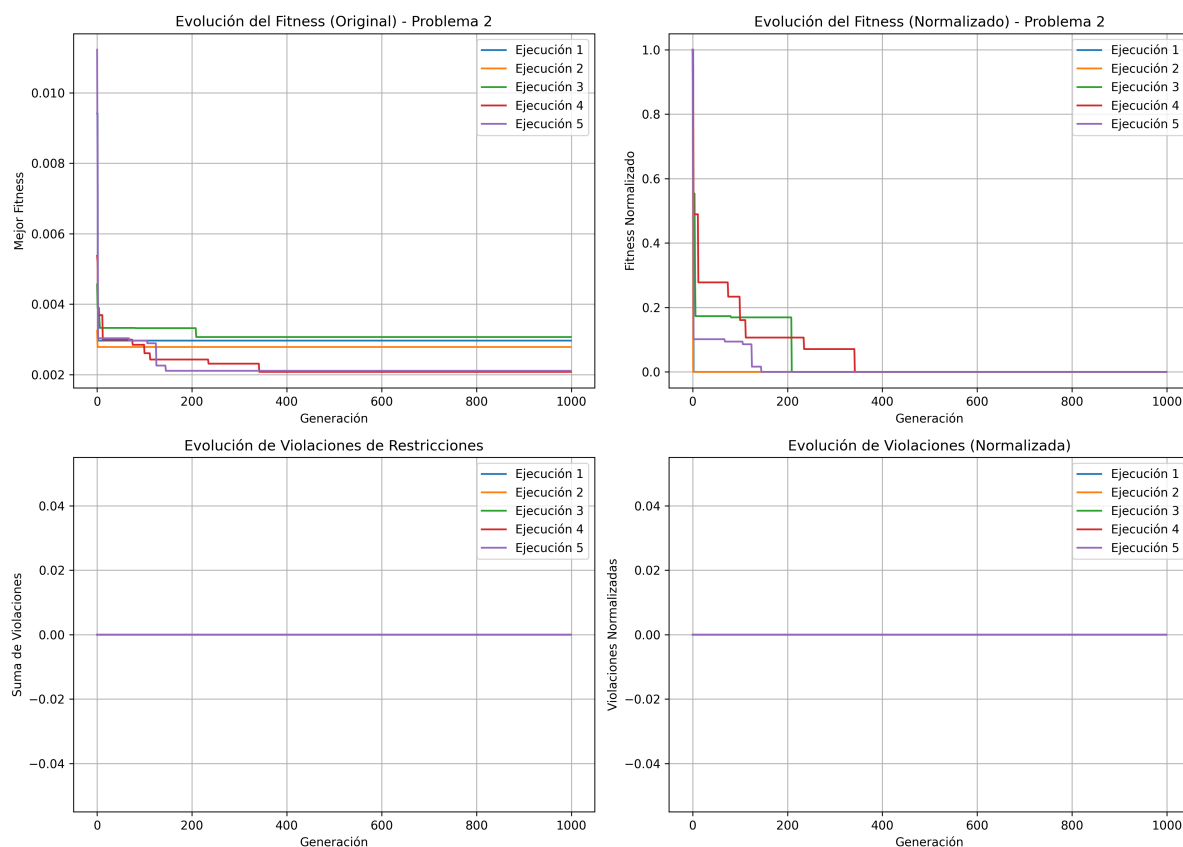


Figura 4.2: Evolución del fitness del Problema 2

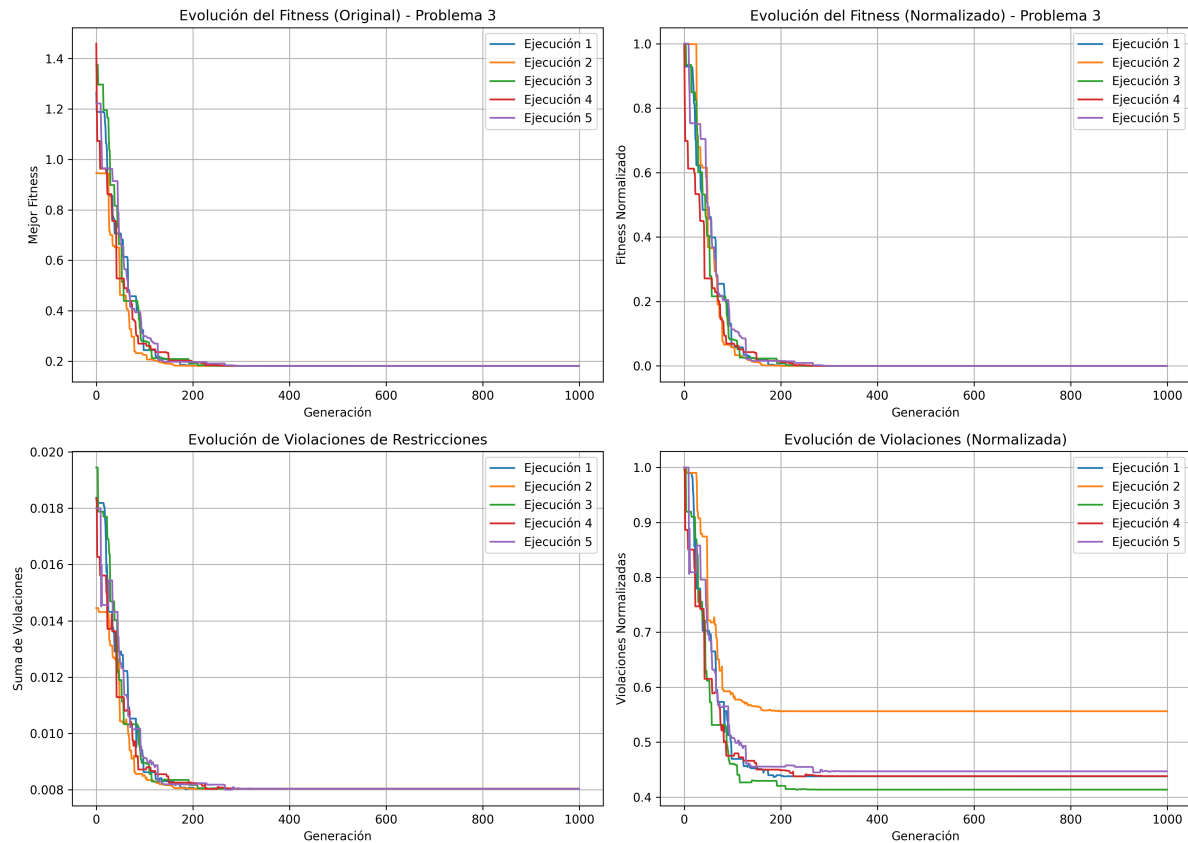


Figura 4.3: Evolución del fitness del Problema 3

## 4.2 Discusión Resultados

- **Eficacia del Algoritmo:** Los indicadores globales extraídos de los CSV demuestran que el algoritmo genético es capaz de acercarse a la solución óptima, manteniendo una evolución progresiva y consistente en la reducción del valor de fitness.
- **Diversidad y Convergencia:** La aplicación de operadores de selección, cruzamiento y mutación, junto con el mecanismo de elitismo, garantiza un equilibrio entre la exploración y la explotación del espacio de búsqueda. Esto se refleja en la baja variabilidad entre corridas, lo que es un indicativo de la estabilidad del proceso.
- **Potencial de Adaptación:** La estructura modular y la robustez mostrada por los resultados permiten considerar la posibilidad de aplicar este marco a otros problemas de optimización, incluso aquellos con mayores dimensiones o con funciones objetivo de mayor complejidad.



## Capítulo 5

# Implementación

El proyecto se ha desarrollado siguiendo una arquitectura modular<sup>1</sup> que permite separar claramente las distintas funcionalidades y facilita tanto el mantenimiento como la ampliación futura. A continuación, se detallan los principales componentes y cómo se integran en el sistema:

## 5.1 Funciones Objetivo

### 5.1.1. Descripción

Las funciones objetivo definen el problema a optimizar. En este proyecto se incluyen dos funciones:

- **Tasa de retorno del portafolio:** Una función que nos permite calcular la ganancia esperada de un portafolio de inversión considerando las ganancias individuales de cada instrumento.
- **Varianza del portafolio:** Una función que nos permite calcular el riesgo de un portafolio de inversión considerando la covarianza de nuestros instrumentos de inversión.

### 5.1.2. Implementación

Se abordaron tres problemas de inversión, cada uno con sus respectivas limitaciones. En los tres casos se establecieron límites para todos los instrumentos de inversión en el intervalo  $[0, 10]$ .

En el primer problema, se buscó maximizar el beneficio del portafolio, es decir, maximizar la tasa de retorno. En el segundo problema, el objetivo fue minimizar la varianza del portafolio, manteniendo una ganancia mínima del 35%; por lo tanto, se incorporó esta restricción al modelo. Finalmente, en el tercer problema se volvió a buscar la maximización de la tasa de retorno del portafolio, pero añadiendo la restricción de considerar un riesgo menor al 20%.

## 5.2 Módulos del Algoritmo Genético

El núcleo del algoritmo genético se distribuye en varios módulos:

### 5.2.1. Inicialización

**Función:** `initialize_population` **Ubicación:** `libs/auxiliaries_functions.py` **Descripción:** Genera la población inicial de manera uniforme en el espacio de búsqueda.

<sup>1</sup>Se puede encontrar el código perteneciente a cada función dentro de

### 5.2.2. Selección

**Función:** `vectorized_tournament_selection` **Ubicación:** `libs/selection.py` **Descripción:** Se usa un enfoque de torneos para la selección de padres, empleando operaciones vectorizadas con NumPy.

### 5.2.3. Cruzamiento

**Funciones:** `sbx_crossover`, `sbx_crossover_with_boundaries` **Ubicación:** `libs/crossover.py` **Descripción:** Implementa el operador SBX (Simulated Binary Crossover) con y sin control de límites.

### 5.2.4. Mutación

**Funciones:** `polynomial_mutation`, `polynomial_mutation_with_boundaries` **Ubicación:** `libs/mutation.py` **Descripción:** Aplica mutación polinomial, con control opcional de límites para mantener la viabilidad de las soluciones.

### 5.2.5. Aplicación de Restricciones

**Función:** `evaluate_individuals_with_constraints` **Ubicación:** `libs/auxiliaries_functions.py` **Descripción:** Verifica el cumplimiento de las restricciones impuestas por el problema. Cuando se violan restricciones, se aplica un mecanismo de penalización para ajustar el valor del fitness, de modo que las soluciones inviables tengan menor probabilidad de ser seleccionadas.

### 5.2.6. Ejecución del Algoritmo

**Función:** `genetic_algorithm` **Ubicación:** `AG.py` **Descripción:** Gestiona el ciclo evolutivo completo del algoritmo genético.

## 5.3 Visualización y Almacenamiento

### 5.3.1. Visualización

**Módulo:** `libs/plot.py` **Funciones:** `plot_evolution_fitness`, `plot_surface_3d` **Descripción:** Permite analizar la evolución del fitness y visualizar la superficie de las funciones objetivo.

### 5.3.2. Almacenamiento

**Estructura de Carpetas:**

- Directorio `outputs` organizado en subcarpetas por problema.
- Historiales en archivos CSV con datos de fitness y variables.
- Resúmenes estadísticos de cada corrida.

**Integración:** `main_script.py` ejecuta el algoritmo para cada función definida en `AG_confs.py`.

**Escalabilidad:** La arquitectura modular permite agregar nuevas funciones objetivo y modificar operadores genéticos sin afectar la estructura base.

## Capítulo 6

# Conclusiones

Los resultados muestran que, bajo restricciones realistas como límites de inversión por activo y control del riesgo, es posible construir carteras bien diversificadas con alto potencial de rendimiento. La optimización permitió alinear expectativas de ganancia con distintos niveles de tolerancia al riesgo, haciendo uso eficiente de la información disponible.

El análisis confirma que considerar covarianzas y varianzas, además de los rendimientos esperados, es clave para evitar decisiones miopes. Así, se demuestra el valor práctico de los modelos de optimización en la construcción de portafolios robustos y alineados con objetivos claros.

## Apéndice A

# GitHub

Se anexa un enlace al repositorio donde se encuentran los codigos y una README que examnde la información aqui mostrada: *Click aqui para abrir el enlace al repositorio* funcion

## Apéndice B

# Scripts

### B.1 Archivo main.py

```
1 import os
2 import pandas as pd
3 from AG_confs import *
4
5 from AG import genetic_algorithm
6 from libs.plot import *
7
8 def main():
9     # Crear carpetas de salida generales
10    os.makedirs("outputs", exist_ok=True)
11
12    for func_key, func_data in FUNCTIONS.items():
13        f_obj = func_data["func"]
14        lb = func_data["lb"]
15        ub = func_data["ub"]
16        func_name = func_data["name"]
17        num_runs = func_data["num_runs"]
18
19        # Carpetas específicas de cada función
20        func_folder = f"outputs/{func_key}"
21        os.makedirs(func_folder, exist_ok=True)
22        hist_folder = os.path.join(func_folder, "historiales")
23        res_folder = os.path.join(func_folder, "resúmenes")
24        os.makedirs(hist_folder, exist_ok=True)
25        os.makedirs(res_folder, exist_ok=True)
26
27        print(f"\n=====")
28        print(f"  FUNCION: {func_name}")
29        print(f"=====")
30
31        all_runs_history = []
32        best_solutions_all_runs = [] # Guardaremos los mejores individuos (x1,
33                                   # x2) de cada corrida
34        best_values_across_runs = [] # Guardaremos el best_val (fitness) de cada
35                                     # corrida
36
37        for run in range(num_runs):
38            print(f"\nEjecucion {run+1}/{num_runs}")
39
40            (best_sol, best_val,
41             worst_sol, worst_val,
42             avg_sol, avg_val,
43             std_val,
44             best_fitness_history,
45             best_x1_history,
46             best_x2_history,
47             population_final,
48             fitness_final,
```

```
47     best_solutions_over_time) = genetic_algorithm(
48         f_obj, lb, ub,
49         pop_size=POP_SIZE,
50         num_generations=NUM_GENERATIONS,
51         tournament_size=TOURNAMENT_SIZE,
52         crossover_prob=CROSSOVER_PROB,
53         eta_c=ETA_C,
54         mutation_prob=MUTATION_PROB,
55         eta_mut=ETA_MUT
56     )
57
58     # 1) Guardar historial
59     df_historial = pd.DataFrame({
60         "Generacion": np.arange(1, NUM_GENERATIONS + 1),
61         "Mejor x1": best_x1_history,
62         "Mejor x2": best_x2_history,
63         "Mejor Fitness": best_fitness_history
64     })
65     historial_filename = os.path.join(hist_folder,
66         f"historial_run_{run+1}.csv")
67     df_historial.to_csv(historial_filename, index=False)
68
69     # 2) Guardar resumen de la corrida
70     data_resumen = [
71         ["Mejor", best_sol[0], best_sol[1], best_val],
72         ["Media", avg_sol[0], avg_sol[1], avg_val],
73         ["Peor", worst_sol[0], worst_sol[1], worst_val],
74         ["Desv. estandar", np.nan, np.nan, std_val]
75     ]
76     df_resumen = pd.DataFrame(data_resumen, columns=["Indicador", "x1",
77         "x2", "Fitness"])
78     resumen_filename = os.path.join(res_folder, f"resumen_run_{run+1}.csv")
79     df_resumen.to_csv(resumen_filename, index=False)
80
81     print(df_resumen.to_string(index=False))
82
83     all_runs_history.append(best_fitness_history)
84     best_solutions_all_runs.append(best_sol)
85     best_values_across_runs.append(best_val)
86
87     # =====
88     # RESUMEN GLOBAL DE LAS CORRIDAS
89     # =====
90     best_values_arr = np.array(best_values_across_runs)
91     solutions_arr = np.array(best_solutions_all_runs) # Cada fila: [x1, x2]
92     del mejor individuo de cada corrida
93
94     # Para el "Mejor" y "Peor", buscamos el indice de la corrida con minimo y
95     # maximo fitness
96     min_index = np.argmin(best_values_arr)
97     max_index = np.argmax(best_values_arr)
98
99     data_global = [
100         ["Mejor (Fitness)", solutions_arr[min_index, 0],
101             solutions_arr[min_index, 1], best_values_arr[min_index]],
102         ["Peor (Fitness)", solutions_arr[max_index, 0],
103             solutions_arr[max_index, 1], best_values_arr[max_index]],
104         ["Media", np.mean(solutions_arr[:, 0]), np.mean(solutions_arr[:, 1]),
105             np.mean(best_values_arr)],
106         ["Desv. Estandar", np.std(solutions_arr[:, 0]),
107             np.std(solutions_arr[:, 1]), np.std(best_values_arr)]
108     ]
109     df_global = pd.DataFrame(data_global, columns=["Indicador", "x1", "x2",
110         "Fitness"])
111
112     global_filename = os.path.join(res_folder, "resumen_global_corridas.csv")
113     df_global.to_csv(global_filename, index=False)
114
115     # (Opcional) Graficar evolucion del fitness de todas las corridas
116     plot_evolucion_fitness(all_runs_history, func_key, func_name)
117
118     # (Opcional) Graficar superficie 3D si la funcion es de 2 variables
119     if len(lb) == 2:
```

```
111         plot_surface_3d(f_obj, lb, ub, best_solutions_all_runs, func_key,  
112                        func_name)  
113 if __name__ == "__main__":  
114     main()
```

Listing B.1: Implementación de main.py

## B.2 Archivo AG\_confs.py

```
1 import numpy as np  
2 from libs.functions import langermann, drop_wave  
3 # -----  
4 # Parametros del algoritmo  
5 # -----  
6 POP_SIZE = 100 # Numero de individuos en la poblacion  
7 NUM_GENERATIONS = 200 # Numero de generaciones  
8 NUM_RUNS = 10 # Numero de ejecuciones completas (ciclos)  
9  
10 # Parametros de la funcion de Langermann  
11 a = np.array([3, 5, 2, 1, 7])  
12 b = np.array([5, 2, 1, 4, 9])  
13 c = np.array([1, 2, 5, 2, 3])  
14  
15 # Parametros del torneo  
16 TOURNAMENT_SIZE = 3 # Numero de individuos participantes en cada torneo  
17  
18 # Parametros del cruzamiento SBX  
19 CROSSOVER_PROB = 0.9 # Probabilidad de aplicar cruzamiento  
20 ETA_C = 15 # indice de distribucion para SBX  
21  
22 # Parametros de la mutacion polinomial  
23 MUTATION_PROB = 10.0 / 2 # Probabilidad de mutar cada gen  
24 ETA_MUT = 20 # indice de distribucion para mutacion polinomial  
25  
26 best_solutions_list = []  
27 all_runs_history = [] # Para graficar luego  
28  
29 FUNCTIONS = {  
30     "langermann": {  
31         "func": langermann,  
32         "lb": np.array([0, 0]),  
33         "ub": np.array([10, 10]),  
34         "name": "Langermann",  
35         "num_runs": NUM_RUNS  
36     },  
37     "drop_wave": {  
38         "func": drop_wave,  
39         "lb": np.array([-5.12, -5.12]),  
40         "ub": np.array([5.12, 5.12]),  
41         "name": "Drop-Wave",  
42         "num_runs": NUM_RUNS  
43     }  
44 }
```

Listing B.2: Implementación de AG\_confs.py

## B.3 Archivo AG.py

```
1 from AG_confs import *  
2 from libs.selection import vectorized_tournament_selection  
3 from libs.crossover import sbx_crossover_with_boundaries  
4 from libs.mutation import polynomial_mutation_with_boundaries  
5 from libs.auxiliaries_functions import initialize_population
```

```
6
7
8 # -----
9 # Funcion principal del GA
10 # -----
11 def genetic_algorithm(objective_func, lower_bound, upper_bound,
12                       pop_size=POP_SIZE, num_generations=NUM_GENERATIONS,
13                       tournament_size=TOURNAMENT_SIZE,
14                       crossover_prob=CROSSOVER_PROB, eta_c=ETA_C,
15                       mutation_prob=MUTATION_PROB, eta_mut=ETA_MUT):
16     """
17     Ejecuta el GA para la funcion objetivo dada y retorna:
18     - best_solution, best_value
19     - worst_solution, worst_value
20     - avg_solution, avg_value
21     - std_value (fitness)
22     - best_fitness_history, best_x1_history, best_x2_history
23     - population (final), fitness (final)
24     - best_solutions_over_time (para animaciones)
25     """
26     num_variables = len(lower_bound)
27
28     # 1) Inicializar poblacion
29     population = initialize_population(pop_size, num_variables, lower_bound,
30                                     upper_bound)
31     fitness = np.array([objective_func(ind) for ind in population])
32
33     best_fitness_history = []
34     best_x1_history = []
35     best_x2_history = []
36
37     # Para animacion: almacenamos el mejor (x1, x2) en cada generacion
38     best_solutions_over_time = np.zeros((num_generations, num_variables))
39
40     for gen in range(num_generations):
41         # Elitismo: guardar el mejor de la generacion actual
42         best_index = np.argmin(fitness)
43         best_fitness = fitness[best_index]
44         elite = population[best_index].copy()
45
46         best_fitness_history.append(best_fitness)
47         best_x1_history.append(elite[0])
48         best_x2_history.append(elite[1])
49         best_solutions_over_time[gen, :] = elite
50
51         new_population = []
52
53         # Numero de padres necesarios (2 por cada par a generar)
54         num_parents_needed = 2 * (pop_size - 1)
55         winners, _ = vectorized_tournament_selection(fitness, num_parents_needed,
56                                                    tournament_size,
57                                                    len(population),
58                                                    unique_in_column=True,
59                                                    unique_in_row=False)
60
61         # Generar un valor global para el crossover y otro para la mutacion (para
62         # toda la generacion)
63         global_u = np.random.rand()
64         global_r = np.random.rand()
65
66         # Generar nueva poblacion
67         for i in range(0, len(winners), 2):
68             parent1 = population[winners[i]].copy()
69             if i + 1 < len(winners):
70                 parent2 = population[winners[i+1]].copy()
71             else:
72                 parent2 = parent1.copy()
73
74             # Cruzamiento SBX usando el mismo u para todas las variables del cruce
75             child1, child2 = sbx_crossover_with_boundaries(
76                 parent1, parent2, lower_bound, upper_bound,
77                 eta_c, crossover_prob, use_global_u=True, global_u=global_u
78             )
```



```
75     # Mutacion polinomial usando el mismo r para todas las variables del
76     individuo
77     child1 = polynomial_mutation_with_boundaries(
78         child1, lower_bound, upper_bound,
79         mutation_prob, eta_mut, use_global_r=True, global_r=global_r
80     )
81     child2 = polynomial_mutation_with_boundaries(
82         child2, lower_bound, upper_bound,
83         mutation_prob, eta_mut, use_global_r=True, global_r=global_r
84     )
85     new_population.append(child1)
86     if len(new_population) < pop_size - 1:
87         new_population.append(child2)
88
89     # Convertir a array y evaluar el fitness de la nueva poblacion
90     new_population = np.array(new_population)
91     new_fitness = np.array([objective_func(ind) for ind in new_population])
92
93     # Incorporar el individuo elite (elitismo)
94     new_population = np.vstack([new_population, elite])
95     new_fitness = np.append(new_fitness, best_fitness)
96
97     # Actualizar la poblacion y su fitness para la siguiente generacion
98     population = new_population.copy()
99     fitness = new_fitness.copy()
100
101     # Calcular estadisticas finales
102     best_index = np.argmin(fitness)
103     worst_index = np.argmax(fitness)
104     best_solution = population[best_index]
105     best_value = fitness[best_index]
106     worst_solution = population[worst_index]
107     worst_value = fitness[worst_index]
108     avg_solution = np.mean(population, axis=0)
109     avg_value = np.mean(fitness)
110     std_value = np.std(fitness)
111
112     return (best_solution, best_value,
113            worst_solution, worst_value,
114            avg_solution, avg_value,
115            std_value,
116            best_fitness_history,
117            best_x1_history,
118            best_x2_history,
119            population,
120            fitness,
121            best_solutions_over_time)
```

Listing B.3: Implementación de AG.py

## B.4 Archivo selection.py

```
1 import numpy as np
2
3 def vectorized_tournament_selection(fitness, num_tournaments, tournament_size,
4                                   pop_size,
5                                   unique_in_column=True, unique_in_row=False):
6     """
7     Genera una matriz de torneos de forma vectorizada y retorna, para cada torneo,
8     el indice del individuo ganador (el de menor fitness).
9
10    Args:
11    - fitness: array con los fitness de la poblacion (longitud = pop_size).
12    - num_tournaments: numero de torneos a realizar (por ejemplo, el numero total
13      de selecciones de padres requeridas en la generacion).
14    - tournament_size: numero de individuos que participan en cada torneo.
15    - pop_size: tamaño de la poblacion.
```

```
15 - unique_in_column: si True, para cada posicion (columna) se eligen
    candidatos sin
16                                     repeticion entre torneos.
17 - unique_in_row: si True, en cada torneo (fila) los candidatos seran unicos.
18   (Por defecto se permite repetir en la fila).
19
20 Returns:
21 - winners: array de indices ganadores (uno por torneo).
22 - tournament_matrix: la matriz de candidatos (de tamaño [num_tournaments x
    tournament_size]).
23
24 if unique_in_row:
25     # Para cada torneo (fila), muestreamos sin reemplazo (cada fila es unica)
26     tournament_matrix = np.array([np.random.choice(pop_size,
27                                     size=tournament_size, replace=False)
28                                     for _ in range(num_tournaments)])
29 else:
30     # Permitir repeticion en la fila, pero controlar la no repeticion en cada
    columna
31     if unique_in_column:
32         # Para cada columna, se genera una permutacion de los indices (o se
    usan numeros aleatorios sin repeticion)
33         # Siempre que num_tournaments <= pop_size.
34         if num_tournaments > pop_size:
35             # Si se requieren mas torneos que individuos, se hace sin la
    restriccion por columna.
36             tournament_matrix = np.random.randint(0, pop_size,
37                                                     size=(num_tournaments, tournament_size))
38         else:
39             cols = []
40             for j in range(tournament_size):
41                 # Para la columna j, se toman num_tournaments indices sin
    repeticion
42                 perm = np.random.permutation(pop_size)
43                 cols.append(perm[:num_tournaments])
44                 tournament_matrix = np.column_stack(cols)
45             else:
46                 # Sin restricciones, se muestrea con reemplazo para cada candidato.
47                 tournament_matrix = np.random.randint(0, pop_size,
48                                                         size=(num_tournaments, tournament_size))
49
50 # Para cada torneo (fila de la matriz), se selecciona el candidato con el
    menor fitness.
51 winners = []
52 for row in tournament_matrix:
53     row_fitness = fitness[row]
54     winner_index = row[np.argmin(row_fitness)]
55     winners.append(winner_index)
56 winners = np.array(winners)
57 return winners, tournament_matrix
```

Listing B.4: Implementación de selection.py

## B.5 Archivo crossover.py

```
1 import numpy as np
2
3 def sbx_crossover(parent1, parent2, lower_bound, upper_bound, eta, crossover_prob):
4     """Realiza el cruzamiento SBX para dos padres y devuelve dos hijos."""
5     child1 = np.empty_like(parent1)
6     child2 = np.empty_like(parent2)
7
8     if np.random.rand() <= crossover_prob:
9         for i in range(len(parent1)):
10             u = np.random.rand()
11             if u <= 0.5:
12                 beta = (2*u)**(1/(eta+1))
13             else:
14                 beta = (1/(2*(1-u)))**1/(eta+1))
```

```
15
16     # Genera los dos hijos
17     child1[i] = 0.5*((1+beta)*parent1[i] + (1-beta)*parent2[i])
18     child2[i] = 0.5*((1-beta)*parent1[i] + (1+beta)*parent2[i])
19
20     # Asegurar que los hijos esten dentro de los limites
21     child1[i] = np.clip(child1[i], lower_bound[i], upper_bound[i])
22     child2[i] = np.clip(child2[i], lower_bound[i], upper_bound[i])
23
24     else:
25         child1 = parent1.copy()
26         child2 = parent2.copy()
27
28     return child1, child2
29
30 def sbx_crossover_with_boundaries(parent1, parent2, lower_bound, upper_bound,
31                                  eta, crossover_prob, use_global_u=False,
32                                  global_u=None):
33
34     """
35     Realiza el cruzamiento SBX con limites, usando formulas que ajustan beta en
36     funcion
37     de la cercania a las fronteras. Permite usar un unico 'u' global para todos
38     los individuos
39     de la generacion o, de forma estandar, un 'u' distinto por cada gen.
40
41     Args:
42     - parent1, parent2: arrays con los padres.
43     - lower_bound, upper_bound: arrays con los limites inferiores y superiores.
44     - eta: indice de distribucion para SBX.
45     - crossover_prob: probabilidad de aplicar el cruce.
46     - use_global_u: si es True se utilizara el mismo valor de 'u' para todas las
47     variables.
48     - global_u: valor de 'u' que se aplicara globalmente (si se proporciona).
49
50     Returns:
51     - child1, child2: arrays con los hijos resultantes.
52     """
53     parent1 = np.asarray(parent1)
54     parent2 = np.asarray(parent2)
55     child1 = np.empty_like(parent1)
56     child2 = np.empty_like(parent2)
57
58     # Si no se realiza el crossover, retornamos copias de los padres.
59     if np.random.rand() > crossover_prob:
60         return parent1.copy(), parent2.copy()
61
62     # Si se quiere usar un 'u' global y no se ha pasado, se genera uno.
63     if use_global_u:
64         if global_u is None:
65             global_u = np.random.rand()
66
67     for i in range(len(parent1)):
68         x1 = parent1[i]
69         x2 = parent2[i]
70         lb = lower_bound[i]
71         ub = upper_bound[i]
72
73         # Aseguramos que x1 sea menor o igual que x2
74         if x1 > x2:
75             x1, x2 = x2, x1
76
77         dist = x2 - x1
78         if dist < 1e-14:
79             child1[i] = x1
80             child2[i] = x2
81             continue
82
83         # Calcular la minima distancia a las fronteras
84         min_val = min(x1 - lb, ub - x2)
85         if min_val < 0:
86             min_val = 0
87
88         beta = 1.0 + (2.0 * min_val / dist)
89         alpha = 2.0 - beta**(-(eta+1))
```

```
84
85     # Si se usa u global, se usa el mismo valor para cada variable
86     if use_global_u:
87         u = global_u
88     else:
89         u = np.random.rand()
90
91     if u <= (1.0 / alpha):
92         betaq = (alpha * u)**(1.0/(eta+1))
93     else:
94         betaq = (1.0 / (2.0 - alpha*u))**(1.0/(eta+1))
95
96     # Calcular los hijos
97     c1 = 0.5 * ((x1 + x2) - betaq * (x2 - x1))
98     c2 = 0.5 * ((x1 + x2) + betaq * (x2 - x1))
99
100    # Ajustar a los limites
101    child1[i] = np.clip(c1, lb, ub)
102    child2[i] = np.clip(c2, lb, ub)
103
104    return child1, child2
```

Listing B.5: Implementación de crossover.py

## B.6 Archivo mutation.py

```
1 import numpy as np
2
3 def polynomial_mutation(child, lower_bound, upper_bound, mutation_prob, eta_mut):
4     """Aplica mutacion polinomial a un hijo."""
5     mutant = child.copy()
6     for i in range(len(child)):
7         if np.random.rand() < mutation_prob:
8             r = np.random.rand()
9             diff = upper_bound[i] - lower_bound[i]
10            if r < 0.5:
11                delta = (2*r)**(1/(eta_mut+1)) - 1
12            else:
13                delta = 1 - (2*(1-r))**(1/(eta_mut+1))
14            mutant[i] = child[i] + delta * diff
15            mutant[i] = np.clip(mutant[i], lower_bound[i], upper_bound[i])
16    return mutant
17
18
19 def polynomial_mutation_with_boundaries(child, lower_bound, upper_bound,
20                                         mutation_prob, eta_mut,
21                                         use_global_r=False, global_r=None):
22     """
23     Aplica mutacion polinomial (con limites) a un vector 'child'.
24     Puede usar un unico 'r' global para todas las variables (si use_global_r=True)
25     o generar un 'r' distinto para cada variable.
26
27     Args:
28     - child : array-like
29         Cromosoma (vector de decision) a mutar.
30     - lower_bound, upper_bound : array-like
31         Limites inferiores y superiores para cada variable.
32     - mutation_prob : float
33         Probabilidad de mutacion (en [0,1]) para cada variable.
34     - eta_mut : float
35         indice de distribucion para la mutacion.
36     - use_global_r : bool
37         Si True, se utiliza un unico valor 'r' para todas las variables.
38     - global_r : float, opcional
39         Valor de 'r' global a usar; si no se proporciona, se genera uno.
40
41     Returns:
42     - mutant : np.ndarray
43         Nuevo vector mutado (manteniendo la dimension de 'child').
```

```
44 """
45 mutant = np.array(child, copy=True, dtype=float)
46 num_vars = len(child)
47
48 # Si se desea usar un 'r' global y no se ha proporcionado, se genera uno una
49 # sola vez.
50 if use_global_r:
51     if global_r is None:
52         global_r = np.random.rand()
53
54 for i in range(num_vars):
55     # Decidir si mutar esta variable
56     if np.random.rand() < mutation_prob:
57         x = mutant[i]
58         xl = lower_bound[i]
59         xu = upper_bound[i]
60
61         # Evitar division por cero si los limites son casi iguales
62         if abs(xu - xl) < 1e-14:
63             continue
64
65         # d = distancia normalizada al limite mas cercano
66         d = min(xu - x, x - xl) / (xu - xl)
67
68         # Elegir r: global o individual para cada variable
69         if use_global_r:
70             r = global_r
71         else:
72             r = np.random.rand()
73
74         nm = eta_mut + 1.0
75
76         # Calcular delta_q segun el valor de r
77         if r < 0.5:
78             bl = 2.0 * r + (1.0 - 2.0 * r) * ((1.0 - d) ** nm)
79             delta_q = (bl ** (1.0 / nm)) - 1.0
80         else:
81             bl = 2.0 * (1.0 - r) + 2.0 * (r - 0.5) * ((1.0 - d) ** nm)
82             delta_q = 1.0 - (bl ** (1.0 / nm))
83
84         # Calcular la nueva posicion y asegurarse que este dentro de los
85         # limites
86         y = x + delta_q * (xu - xl)
87         mutant[i] = np.clip(y, xl, xu)
88
89 return mutant
```

Listing B.6: Implementación de mutation.py

## B.7 Archivo auxiliares\_functions.py

```
1 import numpy as np
2 # -----
3 # Funciones auxiliares del GA
4 # -----
5 def initialize_population(pop_size, num_variables, lower_bound, upper_bound):
6     """Inicializa la poblacion uniformemente en el espacio de busqueda."""
7     return np.random.uniform(low=lower_bound, high=upper_bound, size=(pop_size,
8                                num_variables))
```

Listing B.7: Implementación de auxiliares\_functions.py

## Apéndice C

# Registro de indicadores completo.

## C.1 Problema 01

### C.1.1. Resúmenes

Cuadro C.1: Archivo: resumen\_run\_1

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.1147	0.4	0.3476	0.1435	0.0	-0.6846	0.0437
Media	0.0	0.2109	0.4	0.2012	0.1946	0.0	-0.6502	0.0526
Peor	0.0	0.2101	0.4	0.2409	0.2355	0.0	0.0307	0.0515

Cuadro C.2: Archivo: resumen\_run\_2

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0001	0.0969	0.4	0.3368	0.1476	0.0211	-0.6827	0.0432
Media	0.0	0.2073	0.4	0.2195	0.1565	0.014	-0.6369	0.0514
Peor	0.0	0.2095	0.4	0.389	0.1581	0.0142	2.1541	0.047

Cuadro C.3: Archivo: resumen\_run\_3

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.0968	0.4	0.1406	0.3603	0.0051	-0.6774	0.0533
Media	0.0	0.2175	0.4	0.0735	0.3035	0.0	-0.6078	0.0615
Peor	0.0	0.0935	0.4	0.0706	0.2653	0.0009	2.2822	0.0537

Cuadro C.4: Archivo: resumen\_run\_4

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.0685	0.4	0.4	0.1327	0.0	-0.6897	0.0428
Media	0.0	0.1078	0.4	0.4	0.1028	0.0	-0.6165	0.0426
Peor	0.0	0.1871	0.4	0.4	0.1779	0.0	1.9611	0.0468

Cuadro C.5: Archivo: resumen\_run\_5

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.0	0.4	0.4	0.2024	0.0	-0.6925	0.0456
Media	0.0	0.0	0.4	0.3996	0.1941	0.0	-0.5536	0.0452
Peor	0.0	0.0	0.4	0.3995	0.0405	0.0	1.9406	0.0392

## C.2 Problema 02

### C.2.1. Resúmenes

Cuadro C.6: Archivo: resumen\_run\_1

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.1946	0.1617	0.0308	0.2053	0.2287	0.1787	0.003	-0.3991
Media	0.1211	0.091	0.0443	0.3862	0.0054	0.3566	0.2358	-0.4093
Peor	0.2849	0.1841	0.067	0.3791	0.2101	0.364	23.9317	-0.5967

Cuadro C.7: Archivo: resumen\_run\_2

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.2052	0.1813	0.0249	0.1685	0.2694	0.1506	0.0028	-0.3955
Media	0.1841	0.3093	0.0	0.0324	0.3981	0.089	0.1389	-0.3928
Peor	0.3114	0.3765	0.0	0.0311	0.3985	0.0864	4.1594	-0.4451

Cuadro C.8: Archivo: resumen\_run\_3

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.2556	0.1245	0.0518	0.234	0.1825	0.1515	0.0031	-0.4016
Media	0.0836	0.0505	0.0381	0.3867	0.3833	0.0518	0.0431	-0.4612
Peor	0.0637	0.0512	0.0315	0.3904	0.3864	0.0081	0.5058	-0.4411

Cuadro C.9: Archivo: resumen\_run\_4

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.2863	0.1552	0.0218	0.239	0.1511	0.1463	0.0021	-0.3772
Media	0.2507	0.0025	0.0001	0.3565	0.0175	0.3622	0.1753	-0.3462
Peor	0.2437	0.0174	0.0024	0.2327	0.0216	0.0214	23.7556	-0.1912

Cuadro C.10: Archivo: resumen\_run\_5

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.267	0.162	0.0265	0.1846	0.2143	0.1461	0.0021	-0.3826
Media	0.2924	0.0902	0.0031	0.2219	0.3791	0.0151	0.0195	-0.3893
Peor	0.3097	0.103	0.0019	0.251	0.3824	0.0126	0.3807	-0.4123

## C.3 Problema 03

### C.3.1. Resúmenes

Cuadro C.11: Archivo: resumen\_run\_1

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Media	0.0	0.4	0.4	0.0	0.0	0.203	0.2519	0.1098
Peor	0.0	0.4	0.4	0.0	0.0	0.0886	1.6168	0.1015

Cuadro C.12: Archivo: resumen\_run\_2

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Media	0.0	0.4	0.4	0.0	0.0	0.2135	0.223	0.1107
Peor	0.0	0.4	0.4	0.0	0.0	0.3013	1.0288	0.1187

Cuadro C.13: Archivo: resumen\_run\_3

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Media	0.0	0.4	0.4	0.0	0.0	0.2088	0.1813	0.1103
Peor	0.0	0.4	0.4	0.0	0.0	0.2069	0.1817	0.1102

Cuadro C.14: Archivo: resumen\_run\_4

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Media	0.0	0.4	0.4	0.0	0.0	0.2049	0.2066	0.11
Peor	0.0	0.4	0.4	0.0	0.0	0.145	0.5862	0.1054

Cuadro C.15: Archivo: resumen\_run\_5

Indicador	x1	x2	x3	x4	x5	x6	Fitness	Fitness Secundario
Mejor	0.0	0.4	0.4	0.0	0.0	0.2089	0.1813	0.1103
Media	0.0	0.4	0.4	0.0	0.0	0.2153	0.2585	0.1109
Peor	0.0	0.4	0.4	0.0	0.0	0.3305	1.6513	0.1216