Memoria Práctica 1

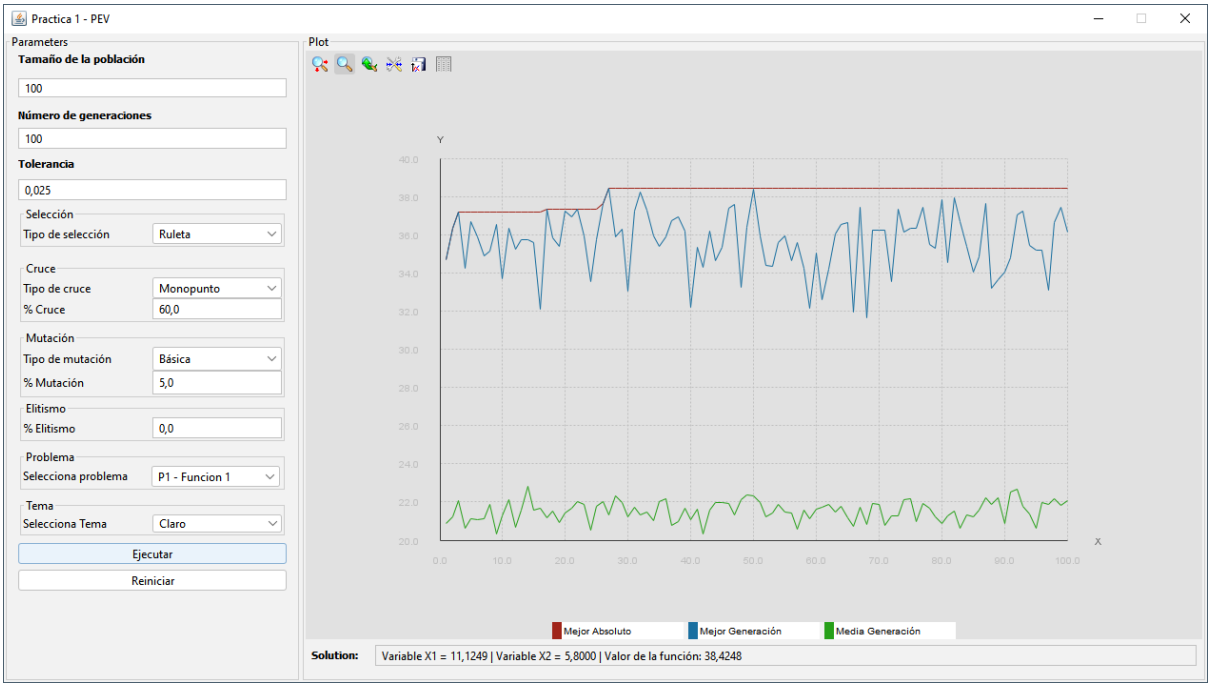


Figura 1. Interfaz del programa

Información general:

- Grupo: 09
- Miembros: Nicolás Rosa Caballero y Javier Meitín Moreno
- Fecha de entrega: 04/03/2023

Reparto de tareas:

Hay algunas clases que hemos programado entre los 2, pero listamos a continuación aquellas donde cada uno ha tenido una participación mayor.

Nicolás Rosa Caballero:

- Arquitectura software del proyecto.
- Algoritmo genético.
- Cromosoma base y gen base.
- Interfaz (MainView y ViewControllerr)
- Algoritmos de selección: Ruleta y Estocástico
- Algoritmo de cruce: BLX-A.
- Cromosoma Función 1
- Parte del cromosoma Función 4A (evaluate)

Javier Meitín Moreno:

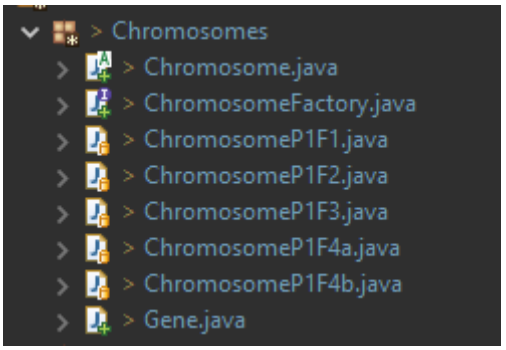
- Algoritmos de selección: Estocástico, torneo (determinístico y probabilístico), truncamiento y restos.
- Algoritmos de cruce: Monopunto, uniforme y aritmético.
- Algoritmo de cruce multipunto (no era obligatorio)
- Algoritmo de mutación
- Cromosomas adaptados a las funciones 2, 3, 4A y 4B.

Organización del proyecto:

Las variables y miembros de campo siguen la nomenclatura Snake Case.
Los métodos siguen la nomenclatura Camel Case.
Los nombres de clase siguen la nomenclatura Pascal Case.
Nuestro proyecto está organizado en 8 módulos:

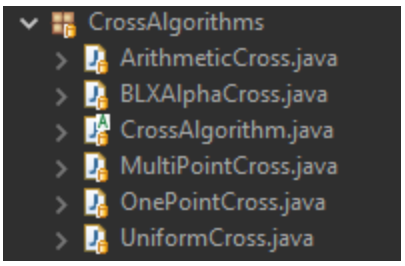
Chromosomes:

Contiene 5 clases llamadas **ChromosomeP1FX**, siendo X el número del ejercicio. Estas clases extienden el template Chromosome y sobrescriben varios métodos para que se adecuen al ejercicio en cuestión. Los cambios más notables son la inicialización de los genes del cromosoma y la función de evaluación. Este paquete también contiene la clase Gene que se usa para codificar los cromosomas. Chromosome es un template <T,U> donde T es el genotipo y U es el fenotipo. Esto permite crear cromosomas que codifican con bits (booleanos), números reales o cualquier otro objeto. Además esto permite también que el fenotipo pueda ser algo distinto a un double, por ejemplo un String o cualquier otro tipo.

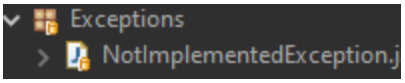


CrossAlgorithms:

Contiene las diversas funciones de cruce. El script base es **CrossAlgorithm** donde se encuentra un método **cross** que genera una nueva población tras el cruzamiento. Cuando se produce un emparejamiento entre 2 individuos llama a un método auxiliar abstracto también llamado **cross** (usamos sobrecarga de argumentos para diferenciarlos). Este método extra es sobreescrito en cada una de las otras clases para que el cruce varíe de manera correspondiente.

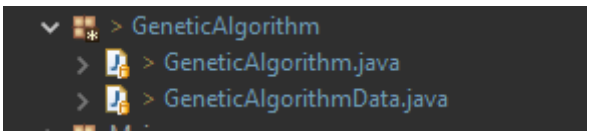


Exceptions:



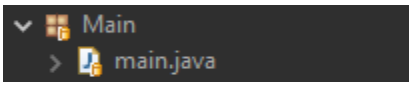
GeneticAlgorithm:

Contiene la clase encargada de ejecutar el algoritmo genético así como su clase de configuración. Esto es debido a que al ser tantos parámetros es más conveniente tenerlos en una clase a parte y pasar un solo objeto a muchos parámetros. Además esto también simplifica el ViewController. Ninguna de estas clases tiene dependencias con la vista o el controlador. GeneticAlgorithm posee los métodos necesarios para poder leer sus datos y ejecutar el algoritmo por partes si llegara a ser necesario, aunque en principio el modelo ejecuta el algoritmo de forma abstracta para el usuario que lo usa (tiene un método run que ejecuta todo el bucle del algoritmo, no hay que montarlo fuera de la clase a baes de llamar a sus funciones). Además GeneticAlgorithm no tiene dependencias directas con los algoritmos de cruce, selección o mutación. Puesto que recibe clases abstractas que contienen dichas funciones, por lo que no sabe en ningún momento qué tipo de selección o cruce está ejecutando. Esta arquitectura permite extender los algoritmos de mutación, cruce o selección sin que esto implique modificaciones a la clase.



Main:

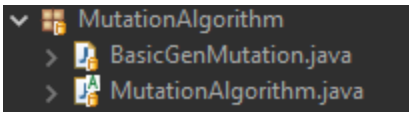
Punto de entrada del software. Contiene el método main que inicializa la vista.



MutationAlgorithm:

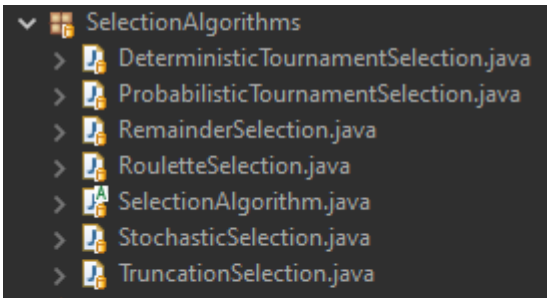
La clases **BasicGenMutation** extiende la clase abstracta **MutationAlgorithm**. Contiene un método **mutate** que recorre en un bucle le array de individuos de una población dada, llamando al método **mutate** de cada cromosoma.

Hemos separado BasicGenMutation y MutationAlgorithm en 2 clases por si en algún futuro necesitamos implementar variaciones del algoritmo de mutación. Es la misma arquitectura de los algoritmos de selección y cruce basada en el patrón *strategy.*



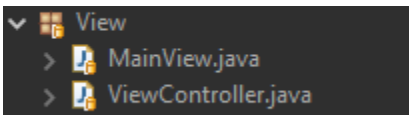
SelectionAlgorithm:

Este módulo contiene los diversos algoritmos de selección especificados en el enunciado. Cada clase extiende la clase abstracta **SelectionAlgorithm** y sobreescribe el método **select**.



View:

Contiene clases para renderizar y controlar la vista. MainView crea el JFrame y gestiona los eventos básicos, se encarga de controlar la validez de los datos introducidos. ViewController es llamado por la vista para actualizar los datos que el usuario introduce. ViewController ejecuta el modelo y actualiza la vista con los datos obtenidos. Es un patrón similar a MVC donde el modelo no conoce la vista ni el controlador, por tanto no tiene dependencias de estas y es posible usarla por separado. El controlador ejecuta el modelo en un hilo a parte, de forma que si este tarda mucho tiempo en ejecutarse la vista no se congela y es posible pararlo en cualquier momento.



UML:

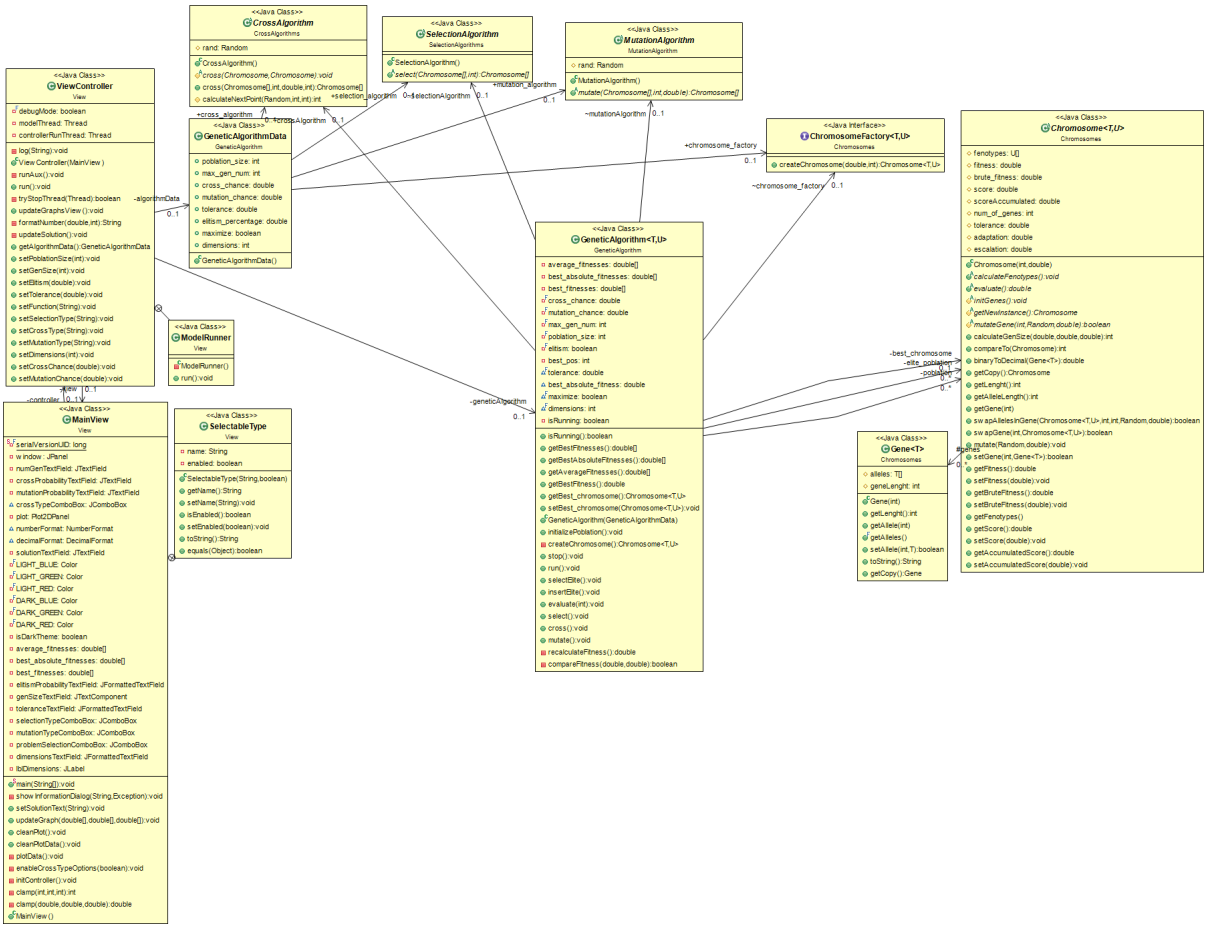


Figura 2. UML Reducido

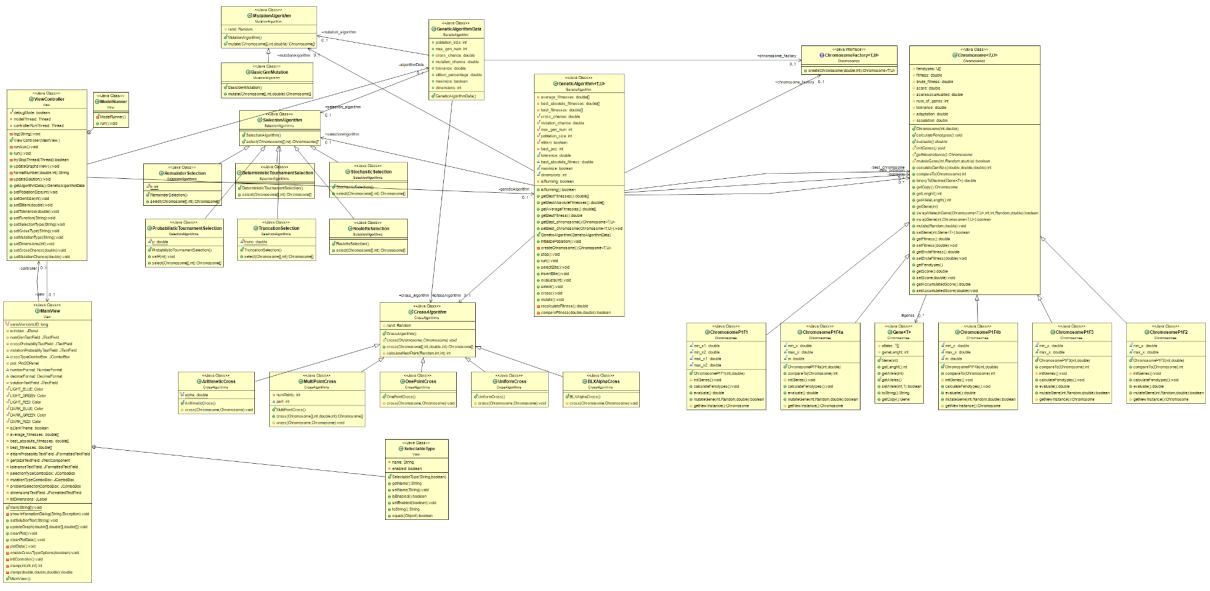


Figura 3. UML Completo

Características de la interfaz:

El panel izquierdo de la interfaz permite configurar una serie de parámetros para el algoritmo tales como el tamaño de la población, el número de generacione, la tolerancia (intervalo mínimo al codificar números con booleanos), el tipo de selección, el tipo de cruce y su probabilidad, el tipo de mutación y su probabilidad, la probabilidad de elitismo, la función y el tema de la interfaz (soporta tema claro y oscuro). Finalmente existe un botón de ejecutar y otro de reiniciar.

El panel derecho contiene la gráfica y debajo de esta un cuadro donde se escribe la solución.

La interfaz garantiza que los datos introducidos son válidos, si el dato introducido no es válido pondrá uno por defecto. Por lo que si escribes 10sfds20 en tamaño de la población la interfaz lo cambiará a 10, que es el valor antes de que se introdujera datos no válidos. Igualmente se controla que no se introduzcan valores negativos, o en los campos de porcentajes que el valor esté entre 0 y 100. Excepcionalmente el campo tolerancia tiene un valor mínimo de 0.001.

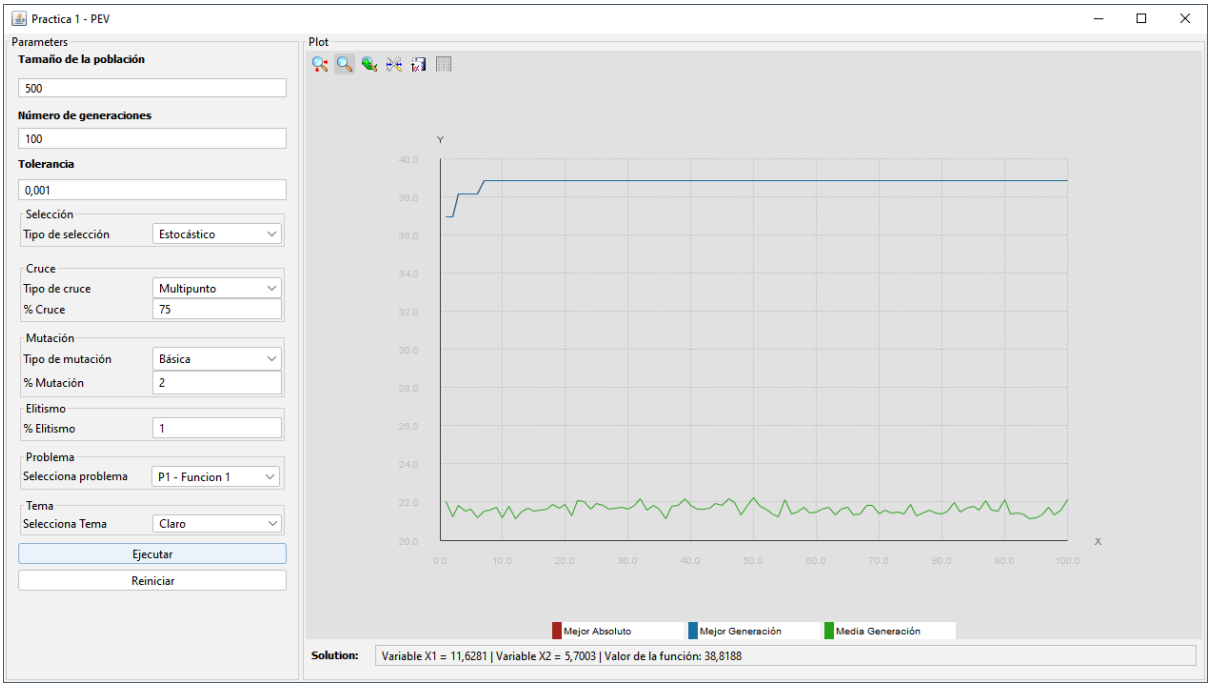
El tamaño de la ventana está por defecto en 1280*720 píxeles. La ventana es escalable, hasta un mínimo de 512 * 256. El panel izquierdo no es scrolleable, por lo que si la ventana es muy pequeña no se podrán ver todas las opciones de configuración.

Los parámetros por defecto en la interfaz son los mismos que los indicados en el guión de la práctica:

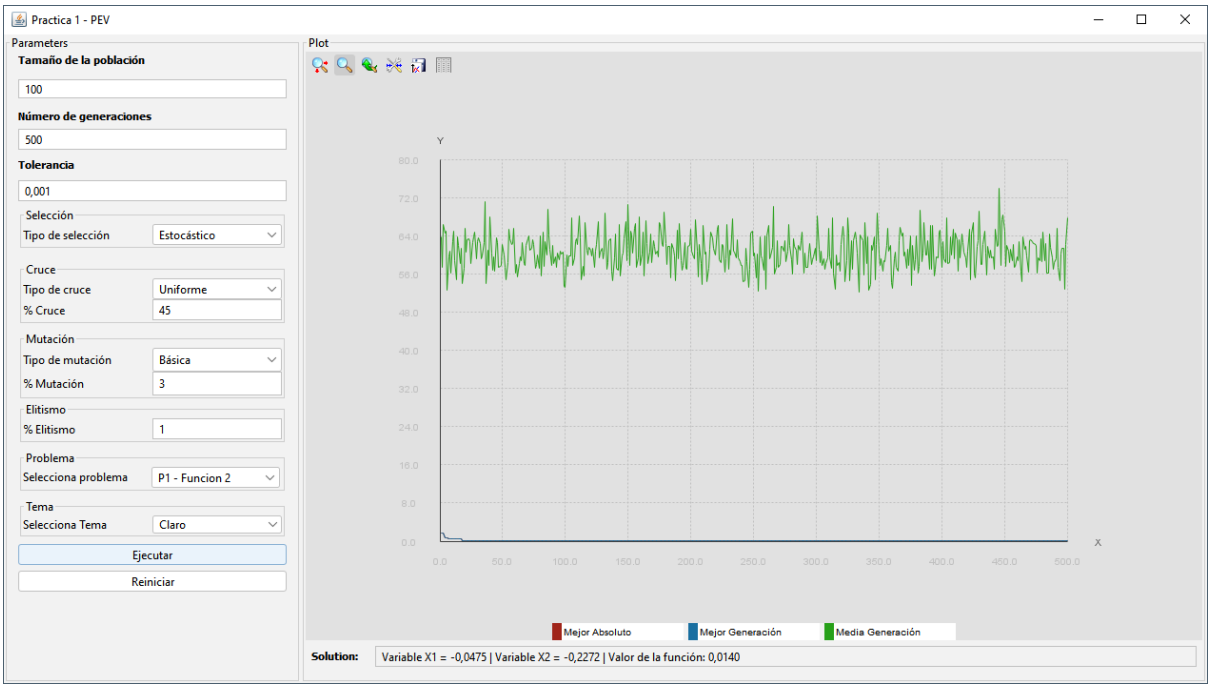
- Tamaño de la población: (100)
- Número de generaciones (100)
- Porcentaje de cruces (60%)
- Porcentaje de mutaciones (5%)
- Precisión para la discretización del intervalo (0.001)
- Método Selección
- Método Cruce
- Método Mutación
- Posibilidad de seleccionar porcentaje de elitismo
- Selección de la dimensión d en la función 4

Gráficas representativas:

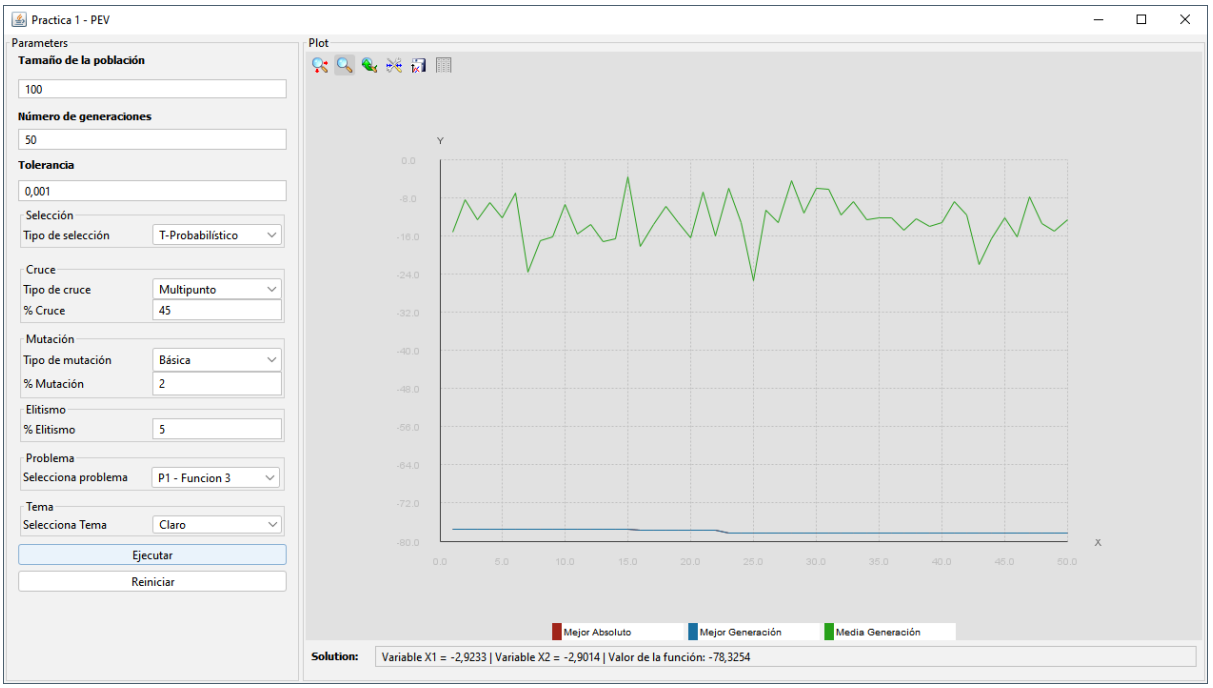
Ejercicio 1:



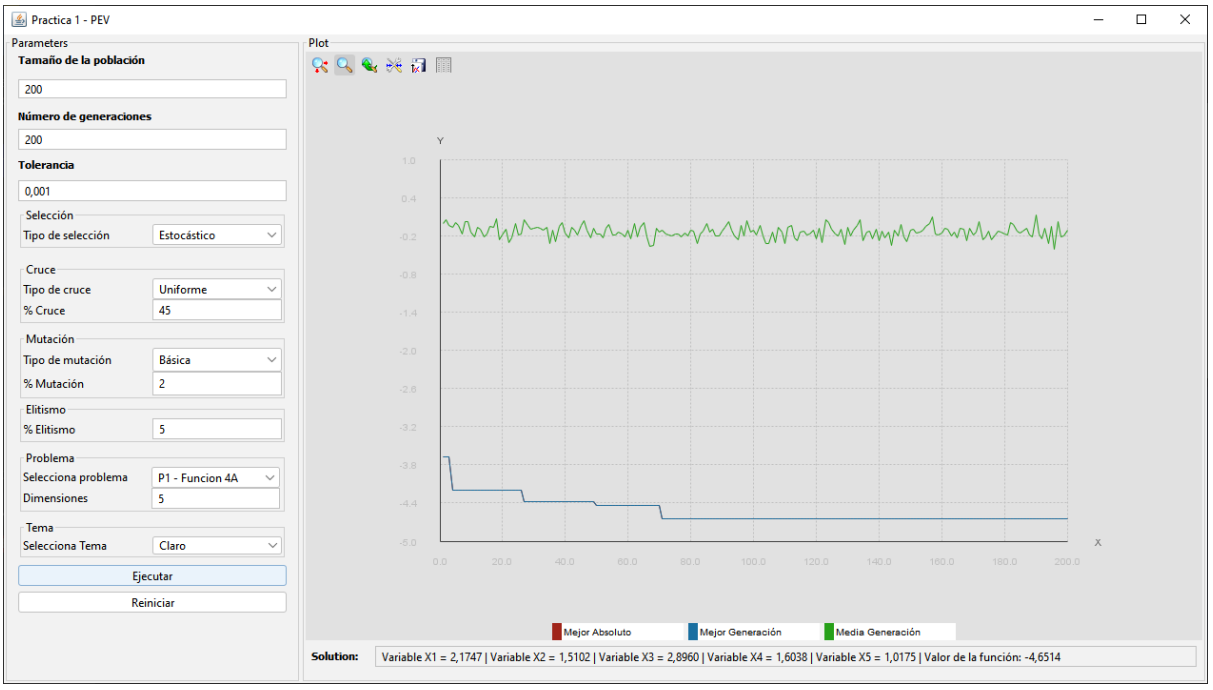
Ejercicio 2:



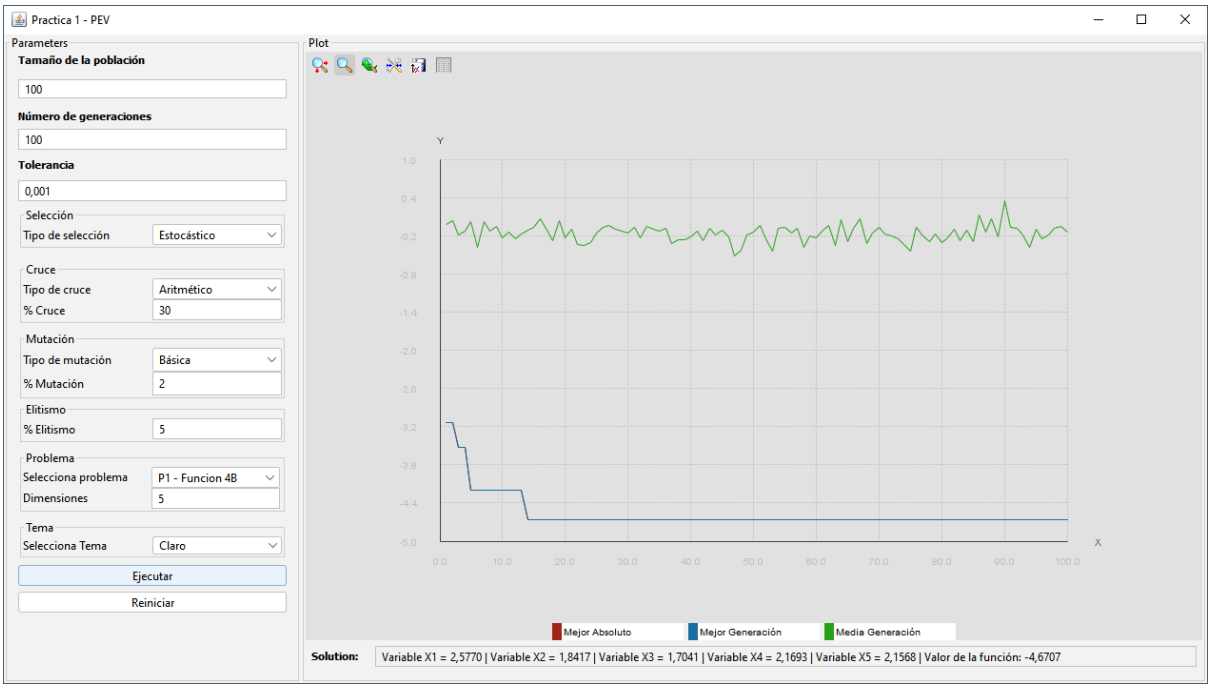
Ejercicio 3:



Ejercicio 4.a:



Ejercicio 4.b:



Conclusiones:

- Se probó a ejecutar el algoritmo en varias funciones probando si es más conveniente aumentar el número de generaciones o el tamaño de la población. Los resultados fueron que, aunque de forma sutil, suele dar mejores resultados aumentar el número de generaciones frente al tamaño de la población. Es más conveniente 200 generaciones y 100 tamaño población que lo contrario.
- El elitismo tiene un impacto considerable en la convergencia. Además, tiene el efecto secundario de que al guardar como mínimo 1 individuo que es el mejor de la generación, la gráfica del mejor de la generación tiende a coincidir con el mejor absoluto.
- El cruce aritmético en el problema 4b parece dar mejores resultados que los demás. El cruce BLX-Alpha parece introducir algo de inestabilidad, aunque a veces da muy buenos resultados (convergencia temprana).
- Las gráficas mostradas son las que obtienen resultados similares con el menor número de individuos y generaciones posibles. Con estas cifras no es apreciable, pero al ejecutar el algoritmo con 1000 individuos y 1000 generaciones, es posible apreciar que la selección estocástica es la que mejores resultados aporta. Así mismo las que peores resultados aporta son Ruleta y Truncamiento. No es una diferencia muy muy notable, pero con dichas cifras llega a ser apreciable tras varias generaciones comparando.
- No conviene usar mutaciones superiores a 5%, introduce bastante ruido, lo cual es apreciable cuando no hay elitismo. Los mejores resultados han sido obtenidos con valores entre 2 y 5%
- El mismo problema sucede con el cruce, aunque depende del problema. Valores entre 30 y 50% suelen dar los mejores resultados, aunque en ocasiones parece funcionar mejor con valores más altos (75% cruce Función 1).

- La tolerancia influye en el tamaño del cromosoma y tiempos de ejecución. Al principio usábamos tolerancia 0,025 porque daba resultados aceptables. Sin embargo, para poder obtener resultados un poco más cercanos a los esperados tuvimos que reducir hasta 0,001. Ese parece ser un punto adecuado entre precisión y rendimiento. Para problemas en los que no se requiere una precisión decimal muy grande aumentaremos dicho valor a 0,025% para poder ejecutar el modelo más veces.
- Con la configuración adecuada el modelo suele converger bastante rápido. Aún así en algunas iteraciones es apreciable como el modelo se mantiene en un máximo estable durante 300 generaciones y luego, de forma súbita, sube el máximo en una generación. En cierto momento hemos considerado implementar la finalización del algoritmo al detectar convergencia, para no estar ejecutando cientos de iteraciones en vano, aunque visto estos resultados hemos decidido que es mejor no hacerlo y ejecutar todas las generaciones hasta el final.