Special Topic:
Genetic Algorithms

Konstantinidis Ioannis

8th Semester

Supervising Professor: Hadjifotinou Katerina

Aristotle University of Thessaloniki

Department of Mathematics

2024

Contents

Introduction

Inspired by genetics and evolutionary theory, Genetic Algorithms are optimization methods that mimic these natural processes. They are among the most popular and widely used optimization algorithms in the fields of artificial intelligence and computer science. We will examine both the classical form of a Genetic Algorithm, as initially proposed by its creator, John Holland, in his book *"Adaptation in Natural and Artificial Systems"* (1975), as well as some of its variations. In the following chapters, we will present examples and explain the functioning of the algorithms and their parameters, starting with analyzing their performance and proceeding with the visualization of more complex optimization problems.

Chapter 1 The Basic Algorithm

In this chapter, we will outline the basic structure of a Genetic Algorithm (GA) and discuss the advantages and disadvantages of GAs. Then, we will take a look at an example of finding the minimum of a function.

A Genetic Algorithm consists of a population of candidate solutions to the problem it is designed to solve, along with an evolutionary process that simulates natural selection. Through this process, the population evolves, with members competing for survival. The criterion for survival is the objective function, for which we seek either a maximum or a minimum. Solutions that perform poorly are less likely to pass their parameters to the next generation, whereas high-performing solutions are more likely to do so. Thus, over successive generations, the population members adopt parameters that increasingly optimize the problem at hand.

In its general form, a GA consists of the following steps (see [3], [5], [7], [8]). We will examine each of these steps:

1) An initial population of candidate solutions
2) An encoding method for the solution parameters
3) A cost/fitness function and a selection method based on it
4) A Crossover method
5) A Mutation method
6) A termination criterion

The **initial population** (IP), consisting of N members, is usually a random set of candidate solutions. However, it is possible for the initial population to be specifically defined if there is prior information about the solution space. For example, the population could be generated

randomly but not uniformly, or it could be derived from another search algorithm, such as another Genetic Algorithm (see examples in Figure 1.1).

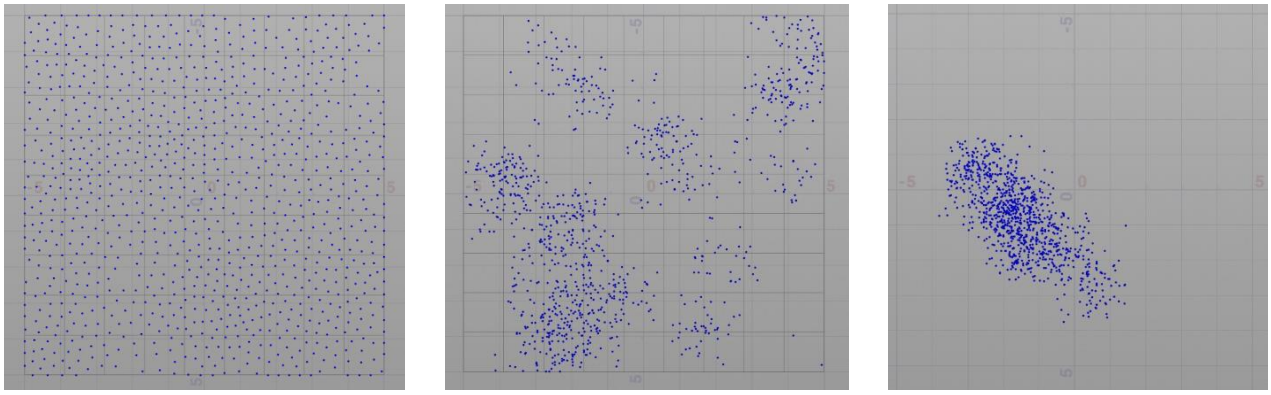

Figure 1.1

Each solution in the initial population is represented through a chosen encoding scheme. This choice is critical, as it directly impacts key stages of the algorithm, as we will see later. A good encoding strikes a balance between exploring the solution space sufficiently and avoiding overanalyzing the problem at the cost of computational time and memory. The most common encoding is binary, where we represent a natural number in its binary form. These representations are referred to as chromosomes, and each bit of the binary representation is called a gene, following biological terminology. For a chromosome of length l, we encode 2^l points. Increasing the chromosome length improves the algorithm's accuracy, though it inevitably consumes more computational resources. For real-valued problems, in the interval $[\alpha, b]$, we linearly map the binary representations x to the interval $[\alpha, b]$, using the function:

$$Remap(x) = \frac{b-a}{2^l - 1}x + a$$

Thus, for example, $x = 0$ is mapped to a, and $x = 2^l - 1$ is mapped to b. Table 1.1 presents an example of an initial population (IP) with $N = 4$ and $0 \leq x \leq 127, \in \mathbb{N}$ (this results from choosing a chromosome length of $l = 7$). In this case, the discretization of the interval $(0,1)$ is done at 128 points, and the remapping is illustrated in Figure 1.2a.

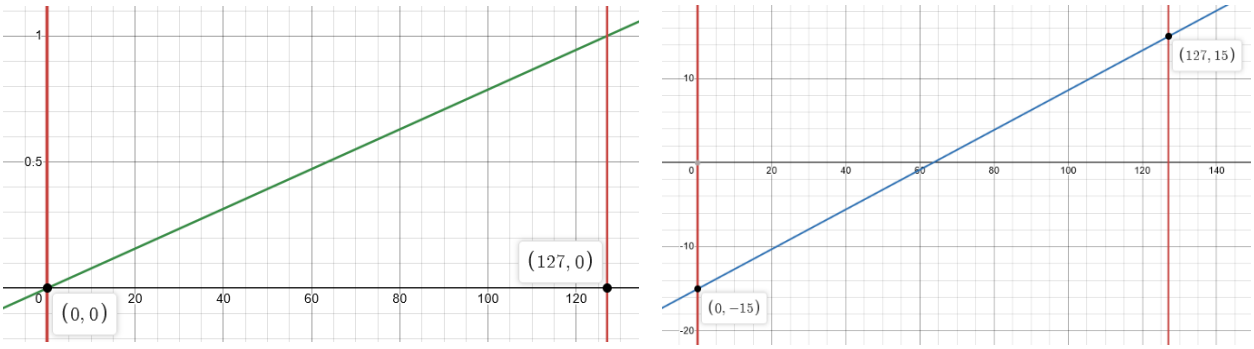

Figure 1.2: Remap of [0,127] a) to [0,1] and b) to [-15,15]

Population Member	x	Binary Encoding	Mapping to [0,1]
1	*5*	*0000101*	*0.03937*
2	*88*	*1011000*	*0.69291*
3	*37*	*0100101*	*0.29134*
4	*9*	*0001001*	*0.07087*

Table 1.1

For the representation of further parameters, K in number, we divide the chromosome length l into K sub-chromosomes of lengths l_i. Obviously $\sum_i l_i = l$, without l_i necessarily being of equal lengths. Depending on the specific requirements of the problem, different l_i can be assigned to each parameter. We can now encode the points from Figure 1.1. For instance, consider a point in $\mathbb{R}^2$ with coordinates in $[-5,5] \times [-5,5]$ and binary encoding where $l_1 = l_2 = 4$ (so the discretization occurs in $[0,15] \times [0,15]$). Thus, the mapping of the encoded point from Table 1.2 to $[-5,5] \times [-5,5]$ is $\left[\frac{(5+5)}{15} \cdot 5 - 5, \frac{(5+5)}{15} \cdot 8 - 5\right] = [-1.6667, 0.3333]$.

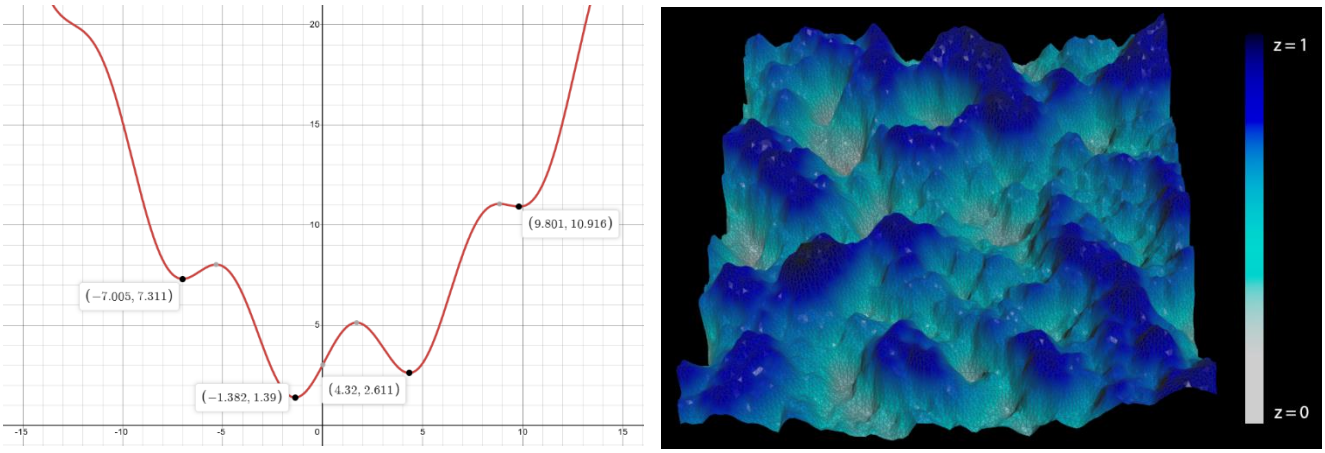

Figure 1.3: a) The minima of f(x)=2 sinx+x +0.1(x-5.5)^2, b) A more complex search space

x	y	Binary x	Δυαδικό y	Binary (x, y)	Mapping to $[-5,5] \times [-5,5]$
5	8	0101	1000	0101|1000 = 01011000	$[-1.6667, 0.3333]$

Table 1.2

The goal of an optimization algorithm is to locally or globally minimize a **cost function**, $Cost: A \rightarrow \mathbb{R}$, (where A is the domain of the problem) or, correspondingly, to maximize a profit function. From now on, we will focus on minimization problems since maximizing a function is equivalent to minimizing its negative, $-Cost$. The range of values that the cost function can take is referred to as the search space or search surface. For example, the function f shown in Figure 1.3 exhibits four local minima and a global minimum for the values of x depicted in the figure. Finding these minima can be achieved using methods that utilize the function's derivatives (such as Gradient Descent). However, in the example on the right, it is less clear where the minima appear. Problems with more than three parameters cannot be fully visualized, and the search space landscape can become quite complex. Furthermore, optimization methods that rely on derivatives tend to exhibit

a rapid increase in complexity as the dimensionality increases due to the need to solve linear systems. Genetic Algorithms (GAs) circumvent these issues.

We need a metric to evaluate how well each solution performs in the algorithm. This metric is known as the **fitness function**. Typically, we define $Fitness = Cost$. Additionally, we define the **average fitness** of the population as follows:

$$f_{avg} = \frac{\sum_i f_i}{N}$$

f_i being the fitness of the i[th] chromosome. In Table 1.3 we see the population from Table 1.1 ($N = 4$), mapped to the interval [-15,15] (see Figure 1.2b), with the $fitness$ function f from figure 1.3a and $l = 7$ thus:

$$Remap(x) = \frac{(15 + 15)}{127}x - 15.$$

Member	x	Binary Encoding	Mapping to $[-15,15]$	$fitness$	$Normalization\ \frac{f_i}{\sum_{i=1}^{4} f_i}$	$\frac{f_{max}+c-f_i}{\sum_{i=1}^{4} If_i}$, $c = 5.5688$
1	5	0000101	-13.8189	21.6035	0.3975	0.1024
2	88	1011000	5.7874	4.8442	0.0891	0.4108
3	37	0100101	-6.2598	7.6162	0.1401	0.3598
4	9	0001001	-12.8740	20.2807	0.3731	0.1268

Table 1.3 (for the last two columns see analysis below)

Crossover is the core of a Genetic Algorithm. This part of the algorithm ensures "sharing" of information between potential solutions. The crossover process (in biology: "recombination") occurs between chromosomes for the creation of the next generation. It happens in two stages:

i. **Selection**
ii. **Crossover**

These processes naturally mimic (i) the process of natural selection: organisms that are better adapted to the environment are more likely to reproduce and (ii) genetic recombination: exchange of genetic material to produce offspring with a combination of parental traits, see Figure 1.4.

Let's consider a trivial case using the example we've been following. This simplification allows us to better observe the individual steps. We want to select four chromosomes from the population to produce four chromosomes for the next generation. By selecting from {5, 88, 37, 9}, and performing a trivial crossover, essentially cloning themselves (e.g., {5} → {5}), we end up with a new, identical population {5, 88, 37, 9}, without introducing any new information.

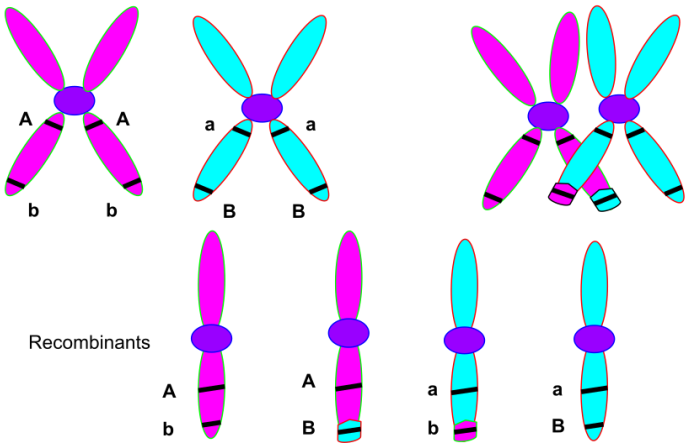

Figure 1.4: Source: https://en.wikipedia.org/wiki/Chromosomal_crossover

Another approach would be to randomly select members of the population, with repetition, such as $\{5,5,88,9\}$. However, this causes the loss of $x = 37$ which had a much lower fitness value compared to $x = 5$ and $x = 9$ (recall we are solving a minimization problem) while 5 appears twice, increasing the likelihood that it will reappear in the future.

Instead of selecting entirely at random, we choose based on the fitness of each chromosome. The basic selection method is called "**roulette selection**" or "fitness-proportional selection". In a maximization problem, each chromosome can have a selection probability $p_i = \frac{f_i}{\sum_i f_i}$, using the normalized fitness values from the 6th column of Table 1.3. For instance, from Table 1.3, we observe that Chromosome 1 has a significantly higher probability of being selected compared to Chromosome 2.

For minimization problems, this method requires some modifications. One simple approach is to apply the same method by setting $fitness = -Cost$ converting the problem into a maximization one. After experimentation, we find that we can add a constant c, to the fitness function, which is analyzed in detail later, resulting in new values: $Inverted\ fitness = If = f_{max} + c - f_i, c > 0$ and new probabilities:

$$p_i = \frac{f_{max} + c - f_i}{\sum_i If_i}$$

The scaling constant c is used to adjust the range of the fitness function values to a desired range. As c increases, asymptotically we will have $p_i \to \frac{C}{N \cdot c} = \frac{1}{N}$, while for $c = 0$, f_{max} will have $p_i = 0$. Generally, c is chosen experimentally, by testing various values and observing the algorithm's behavior. For this reason, c is left as a tunable parameter in the algorithm (large c favors weaker solutions, while small c acts as a form of elitism—see Chapter 5 for more on elitism). You can choose a small absolute value, such as $c = 0.001$, or use a proportion of the range $(f_{max} - f_{min})$ such as $c = 0.01(f_{max} - f_{min})$. It's also possible for c to vary throughout generations.

Figure 1.5 shows how the selection probabilities change as a function of c. For example, when $c = 0$ the probabilities are $p_1 = 0$, $p_2 = 0.0412, p_3 = 0.4362, p_4 = 0.5226$. The red dashed vertical line corresponds to $c = 5.5688$ as given in the last column of Table 1.3, yielding selection probabilities $p_1 = 0{,}1025$, $p_2 = 0.1268, p_3 = 0.3599, p_4 = 0.4509$. or large values of c, we observe that $p_i \to 0.25$.

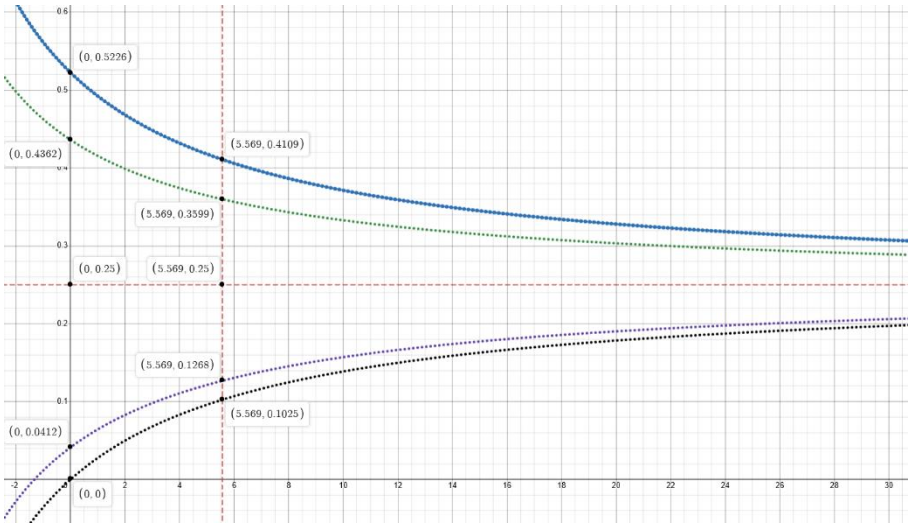

Figure 1.5: Probabilities p_i as a function of c. On color Black: p_1, Blue: p_2 2, Green: p_3, Purple: p_4

Having selected a method for choosing chromosomes that perform well for the problem at hand, we now turn to the crossover process. Let's revisit the earlier trivial example. If crossover were merely cloning, the next generation would consist of the selected chromosomes and because of favoring fittest solutions: $f_{avg} \xrightarrow{G \to \infty} f_{max}$, where G represents the generation of the population (for the initial population, we set $G = 0$). In this case, the algorithm would only explore the chromosomes in the initial population. To discover new chromosomes, we introduce a simple but effective method of exchanging information between them, which closely resembles the process shown in Figure 1.4.

Crossover requires two selected chromosomes. Crossover occurs with a probability p_δ, which allows the possibility for chromosomes to be passed on to the next generation unchanged, maintaining some of the genetic information from the previous generation. When crossover happens, a random point is selected in the binary representation of the chromosomes. At this point, the chromosomes are "split" into two parts, and these parts are swapped between the chromosomes. The resulting two new chromosomes are then passed on to the next generation.

Now, let's assume that the 3rd and 4th members of the population were selected from Table 1.3. In the case where no crossover occurs, the population in the next generation will contain two chromosomes identical to the previous ones. However, if crossover takes place, the result is shown in Table 1.4, along with the main characteristics of the new solutions. For the random breakpoint $J \in \mathbb{N}$ where the chromosomes are split, it holds that $1 \leq J \leq l - 1$ and let $J = 4$, with numbering as follows: $0|_1 1|_2 0|_3 0|_4 1|_5 0|_6 1$.

Member	x	$fitness$	Binary Encoding	Breakpoint at $J = 4$	
3	37	7.6162	0100101	$0100\|_4101$	$\boldsymbol{G = 0}$
4	9	20.2807	0001001	$0001\|_4001$	
			0100 001 → 0001 101 0100 101 → 0001 001		⋮
Member	Binary Encoding	x	Mapping to [-15, 15]	$fitness$	$\boldsymbol{G = 1}$
3’	0100001	33	-7.2047	7.3432	
4’	0001101	13	-11.9291	19.6382	

Table 1.4

By applying the same methodology to the other two selected members, we obtain two additional new members for the second generation. The algorithm’s loop can be repeated until a termination criterion is met. A Genetic Algorithm (GA) may incorporate one or more of the following criteria. Depending on the specific goal of the GA, more specific criteria, tailored to the problem at hand may be used.

- The most common termination criterion is achieving a predefined number of generations, for example $G = 50, 100, 200$.
- Lack of fitness improvement over some generations: $|f_{max}^{G=n} - f_{max}^{G=n-1}| < \varepsilon$ and/or $|f_{avg}^{G=n} - f_{avg}^{G=n-1}| < \varepsilon$ with given ε for a number of consecutive generations (Essentially, the process continues until the current f_{max} stops changing over some generations).

Usually, after crossover, we add one more step: mutation. This step mimics the random changes in the genome caused by errors in natural processes or environmental factors. In the context of a GA, mutation provides the algorithm with the ability to escape local minima and explore gene combinations that would not have arisen through the crossover of existing chromosomes.

Let p_μ represent the mutation probability for each bit. Depending on the problem’s requirements, p_μ is typically quite small (ranging from 0.01 to 0.2). When a bit undergoes mutation, it flips from 0 to 1 or from 1 to 0. Every bit in the population has the same mutation probability p_μ. Suppose the third member of the population undergoes a mutation at the 3rd bit (see Table 1.5):

Member	x	Binary Encoding	Mutation	Binary Encoding	x	Mapping to [-15, 15]	$fitness$
3’	33	0100001	→	0110001	49	-3.4252	5.1003

Table 1.5

It should be noted that here we observe an immediate improvement in fitness (since we are dealing with a minimization problem), this isn’t necessarily the intended result. For example, if the mutation occurred at the 2nd bit, we would get $x = 1$, mapped to -14.7638, with fitness degrading significantly (24.6783). The objective is to explore new combinations, and mutations with low fitness are likely to be discarded by selection in subsequent generations.

Example 1

We have now described the core structure of a Genetic Algorithm (GA). In Program 1, function example1() (see appendix), provides an implementation of this basic algorithm in Python. Initially, we test the program on the function f from Figure 1.3a, using the default parameters:

```
#Global parameters
G = 10  #total generations
N = 8  #number of chromosomes per generation. Even number
L = 9  #binary length
C_MINMAX_FIT_PERCENTAGE = 0.01
Cross_prob = 0.8  #probability of crossover
Mutation_prob = 0.01  #probability of mutation
```

This means that the algorithm runs for 10 generations with 8 chromosomes per generation, 9 bits in each chromosome (binary string), $p_\delta = 0.8$, $p_\mu = 0.01$ and returns the following. Parameter C_MINMAX_FIT_PERCENTAGE determines the value of c for roulette wheel selection as follows (The concept of diversity is explained below):

$$c = C\_MINMAX\_FIT\_PERCENTAGE * (f_{max} - f_{min})$$

```
change algorithm parameters in line 5
function can be changed in line 18
--Generation: 0--
Average fitness:  10.472433478285115
Diversity:  0.46825396825396826
--Generation: 1--
Average fitness:  8.885306200589591
Diversity:  0.45634920634920617
--Generation: 2--
Average fitness:  8.240995127935548
Diversity:  0.376984126984127
--Generation: 3--
Average fitness:  4.486153699432374
Diversity:  0.32142857142857145
--Generation: 4--
Average fitness:  4.948262370175348
Diversity:  0.2738095238095238
--Generation: 5--
Average fitness:  1.4264316671627348
Diversity:  0.11111111111111113
--Generation: 6--
Average fitness:  1.5163329588482815
Diversity:  0.18253968253968253
--Generation: 7--
Average fitness:  1.3897342294565984
Diversity:  0.0
--Generation: 8--
Average fitness:  1.3897342294565984
Diversity:  0.0
--Last Generation: 9--
['001101000', '011101000', '011101000', '011101000', '011101000', '011101000', '011111000',
'011101000']
Average fitness:  2.673483033024084
Diversity:  0.05555555555555566
Best solution:  1.3897342294565984
```

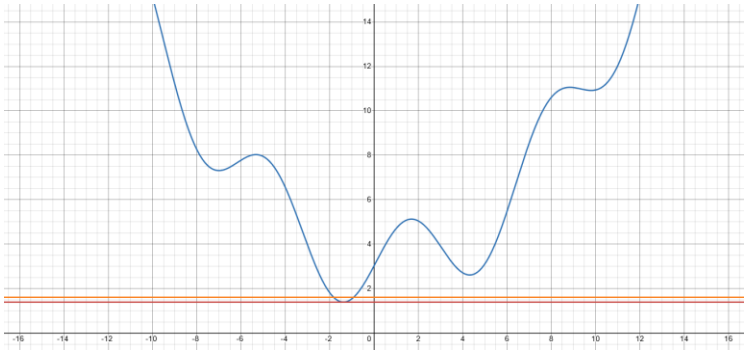

Figure 1.6: For the last generation red indicates optimum fitness and orange f_{avg}

We observe that in just 10 generations, the algorithm effectively identifies the region of the global minimum (see Figure 1.6). While the precision of the solution is not our primary concern, the chromosome 011101000, which produces the lowest $fitness$, corresponds to $x = -1.3796$, which is quite close to the actual value $x = -1.3815$ (with a fitness of 1.38973, the desired global minimum), considering the limitations introduced by the discretization of the search space. Additionally, we notice that f_{avg} generally exhibits a decreasing trend. The behavior of f_{avg} over generations gives us some insight into the algorithm's progression, but it doesn't provide information about the number of distinct solutions in each generation. Thus, we need to introduce the concept of $diversity$ [4]:

$$d_H(a,b) = \sum_i \delta(a_i, b_i)$$

$$\delta(a_i, b_i) = \begin{cases} 1\,, a_i \neq b_i \\ 0\,, a_i = b_i \end{cases}$$

$d_H(a,b)$ is the Hamming distance between two binary representations a, b, that is, the number of pairwise differing bits between chromosomes. For example: $d_H(01110, 01011) = 2$.

As a measure of $diversity$ of the population in generation G, we calculate the normalized pairwise sum of the Hamming distances between the chromosomes in the population:

$$diversity(G) = \frac{2\sum_{i\neq j}\frac{d_H(i,j)}{l}}{N(N-1)} = \frac{2\sum_{i\neq j} d_H(i,j)}{N(N-1)l}$$

Where the terms N, l represent the population size and the length of the binary representation, respectively. $\frac{d_H(i,j)}{l}$ is the normalized Hamming distance, while the denominator results from dividing by $\frac{N(N-1)}{2}$ as we sum over $\binom{N}{2}$ pairs. For instance, consider a population with $N = 3,\ l = 3$ in generation G consisting of the chromosomes “010”,”100”,”110”. In this case, we compute the Hamming distances as:

$\sum_{i\neq j} d_H(i,j) = d_H(010,100) + d_H(010,110) + d_H(100,110) = 2 + 1 + 1 = 4$, and thus:
$diversity(G) = \frac{2*4}{N(N-1)l} == \frac{2*4}{3*3*2} = 0.444$.

In Example 1, we observe that $diversity$ exhibits a declining trend. Notably, during the 7th and 8th generations, the population consisted entirely of the chromosome "011101000", resulting in $diversity = 0$. In the 9th generation, two chromosomes undergo mutation, causing a temporary increase in f_{avg} for that generation.

Advantages and Disadvantages

At this point, it's worth comparing Genetic Algorithms (GAs) with other optimization algorithms (see e.g., [3], [7], [11]). The main advantages of GAs are:

1. **Flexibility in Solution Representation**: GAs offer considerable flexibility in how solutions are represented. As we will see later, GAs are not limited to binary representation. With appropriate encoding, they can tackle a wide range of problems.
2. **Applicability Across Problem Types**: GAs can be applied to discrete values, analytical functions, experimental, or synthetic data. They do not require knowledge of the derivatives or the continuity of the objective function.
3. **Parallel Processing Capability**: Due to their structure, GAs can leverage modern parallel computing architectures. The evaluation of each solution is independent of the fitness of other solutions, meaning these evaluations can be distributed across different processing units. Furthermore, more complex parallelization techniques can be applied to the processes of crossover and mutation as well.
4. **Balance Between Diversification and Intensification**: Depending on parameter settings, a GA can switch between exploring new solutions (diversification) and converging towards a particular solution (intensification). This flexibility allows GAs to explore complex solution spaces and discover solutions without getting trapped in local optima.
5. **Insight Into the Problem Space**: The evolving population over generations provides valuable information about the problem's nature. By analyzing data such as f_{avg}, solution $diversity$, distribution of solutions, convergence behavior and selection methods, we can draw conclusions about the solution space.

Disadvantages:

1. **High Computational Cost**: When the fitness evaluation is computationally expensive, GAs can amplify this issue as they often require multiple fitness evaluations.
2. **Representation Challenges**: The flexibility in solution representation can also be a drawback. If the problem's encoding is not suitable, it may lead to a poor representation of the search space and, consequently, suboptimal solutions.
3. **Tuning Parameters**: Finding the right parameters, such as p_μ, p_δ, N, etc., can be quite complex. Parameters that work well for one problem may not necessarily optimize the algorithm for even similar problems. Additionally, GAs can be highly sensitive to parameter changes. For example, small variations in p_μ may result in slower convergence or convergence to a specific local minimum.
4. **Risk of Local Optima**: Like many optimization algorithms, GAs can get trapped in local minima if there is no mechanism to sufficiently promote diversification. A GA cannot guarantee finding a global optimum.

Chapter 2 Analysis of Genetic Algorithm Parameters

For different parameters (for example, the number of generations G, $p_\delta, p_\mu, ...$), the algorithm showcases significant differences in the speed and accuracy of convergence. The combinations of parameters are numerous, and their optimization is a crucial problem, especially in problems involving many variables. We will begin by analyzing the impact of each parameter on the algorithm. In Program 1, function example2() executes the same Genetic Algorithm as in example1(). However, due to the stochastic nature of the algorithm, it is executed a number of times (variable `reps = 1000` in the program) and we average out the outcomes. Thus, the results we obtain are an overall average performance of the algorithm. The parameter values we will check are the following, corresponding to the variables in Example 1:

```
G_list = [5, 10, 15, 25, 50]  #total generations
N_list = [4, 8, 12, 16, 20]  #number of chromosomes per generation. Even number
L_list = [6, 9, 12, 15, 20]  #binary length
C_prob_list = [1, 0.8, 0.7, 0.5, 0.3]  #probability of crossover
M_prob_list = [0, 0.01, 0.05, 0.1, 0.3]  #probability of mutation
#average results for number of reps
```

The section of the program that controls which parameter we analyze is as follows. By replacing L_list and L with, for example, G_list and G, we can analyze the algorithm with respect to the number of generations instead of the length of the binary representation:

```
for x in L_list:
  global L
  L = x
```

For G:

```
for x in G_list:
  global G
  G = x
```

By running example2() for all parameters, we obtain the results shown in Table 2.1. On the left, we have the set of parameters in the following order: G, N, L, Cross_prob and Mutation_Prob. On the right, we sequentially display the following: the average minimum fitness (min_fit) that the algorithm achieves in one of its generations, the average fitness of the population (avg_fit) in the generation where the algorithm achieves the min_fit, the diversity of the population in the generation where the algorithm achieves the min_fit, and the average generation number where this min_fit occurs. We track information from the generation where the algorithm reaches the min_fit because, often, due to mutations, the algorithm may lose the minimum point in subsequent generations.

5 8 9 0.8 0.01 : [2.35, 9.64, 0.43, 0.63] in 2.23 sec 10 8 9 0.8 0.01 : [2.2, 9.23, 0.39, 1.57] in 4.42 sec 15 8 9 0.8 0.01 : [2.12, 8.96, 0.36, 2.61] in 6.3 sec 25 8 9 0.8 0.01 : [2.01, 8.65, 0.32, 5.12] in 10.57 sec 50 8 9 0.8 0.01 : [1.89, 8.12, 0.28, 12.17] in 22.68 sec
10 4 9 0.8 0.01 : [3.31, 8.74, 0.34, 1.48] in 2.37 sec 10 8 9 0.8 0.01 : [2.2, 9.23, 0.39, 1.57] in 5.87 sec 10 12 9 0.8 0.01 : [1.82, 9.55, 0.41, 1.84] in 9.03 sec 10 16 9 0.8 0.01 : [1.69, 9.81, 0.43, 1.89] in 15.31 sec 10 20 9 0.8 0.01 : [1.58, 9.89, 0.44, 2.04] in 23.31 sec
10 8 6 0.8 0.01 : [2.12, 9.64, 0.41, 1.18] in 5.06 sec 10 8 9 0.8 0.01 : [2.2, 9.23, 0.39, 1.57] in 5.61 sec 10 8 12 0.8 0.01 : [2.25, 9.03, 0.37, 1.78] in 5.23 sec 10 8 15 0.8 0.01 : [2.27, 8.81, 0.37, 1.85] in 4.75 sec 10 8 20 0.8 0.01 : [2.31, 9.02, 0.37, 1.92] in 6.65 sec
10 8 9 1 0.01 : [2.15, 9.12, 0.38, 1.69] in 4.83 sec 10 8 9 0.8 0.01 : [2.2, 9.23, 0.39, 1.57] in 4.48 sec 10 8 9 0.7 0.01 : [2.24, 9.22, 0.39, 1.52] in 4.82 sec 10 8 9 0.5 0.01 : [2.28, 9.22, 0.39, 1.5] in 5.22 sec 10 8 9 0.3 0.01 : [2.36, 9.4, 0.4, 1.45] in 5.19 sec
10 8 9 0.8 0 : [2.39, 9.41, 0.42, 0.91] in 4.7 sec 10 8 9 0.8 0.01 : [2.2, 9.23, 0.39, 1.57] in 4.8 sec 10 8 9 0.8 0.05 : [1.84, 9.22, 0.39, 2.76] in 4.88 sec 10 8 9 0.8 0.1 : [1.66, 9.18, 0.42, 3.13] in 4.86 sec 10 8 9 0.8 0.3 : [1.49, 9.68, 0.49, 3.52] in 5.9 sec

Table 2.1

Let's analyze the results of the table. Starting with parameter G, we observe that by increasing the total number of generations, the algorithm achieves on average lower minimum fitness. This is because the algorithm has more time to search for the minimum of the fitness function. This trend is confirmed by the increase in the average generation where min_fit is achieved. For example, with G=25, we achieve an average minimum fitness of 2.01 in 5.12 generations, whereas with G=50, we achieve 1.89 in 12.17 generations. Furthermore, the decrease in diversity at the generation where min_fit is achieved indicates that the algorithm is converging towards a minimum. However, increasing the number of generations leads to an expected almost-linear increase in the program's execution time. Therefore, increasing G is not always an ideal choice for improving the algorithm's performance.

Regarding the parameter N, we observe similar results in terms of finding the minimum and execution time (likely with a superlinear increase in time instead of linear). By increasing N, we achieve a lower min_fit, but with increased computational cost. Unlike G, increasing N leads to higher diversity at the generation where min_fit is achieved, indicating that the algorithm is better scanning the problem's search space. This is also reflected in the average avg_fit at the generation where min_fit is achieved, which increases as more chromosomes are farther from the minimum point. For the parameter L, we observe that in this particular minimization problem, smaller values of L, such as L=6, are more favorable, achieving a lower minimum on average in fewer generations. The execution time for different values of L doesn't seem to differ significantly, nor

do the average diversity or avg_fit show significant deviations. However, it is worth noting that L affects the solution's precision; therefore, the smallest possible value of L that provides sufficient precision for the problem's requirements should be chosen.

Crossover probability does not appear to significantly affect the algorithm's performance in finding the minimum of the given cost function. Based on the results, we can say that for $p_\delta = 1$, compared to lower values, the algorithm seems to achieve a slightly lower average minimum fitness in slightly more generations. On the other hand, the mutation probability has a more significant impact on the algorithm's performance. Higher values of p_μ tend to favor finding a lower average minimum fitness, though over a larger number of generations. For example, for $p_\mu = 0.01$, the algorithm achieves an average minimum fitness of 2.2 in 1.57 generations, while for $p_\mu = 0.1$, it achieves 1.66 in 3.13 generations. A higher mutation probability also slightly increases execution time, as it makes it more likely that the mutation function will be executed, which is evident from the results.

Based on the forementioned observations we can make a selection of parameters that may be optimal for this specific problem. We choose N=12, L=6, Cross_prob=1, Mutation_prob=0.3 and execute the example2() function for these parameters, looping over G:

```
5 12 6 1 0.3 : [1.56, 10.11, 0.49, 0.94] in 5.18 sec
10 12 6 1 0.3 : [1.46, 10.06, 0.49, 2.33] in 11.25 sec
15 12 6 1 0.3 : [1.44, 10.04, 0.49, 3.24] in 15.36 sec
25 12 6 1 0.3 : [1.43, 10.0, 0.49, 4.43] in 29.15 sec
50 12 6 1 0.3 : [1.43, 10.01, 0.49, 4.95] in 52.64 sec
```

Initially, we observe that increasing the number of generations beyond G=15 offers little to no additional benefit to the algorithm apart from added computational cost. The results are already quite satisfactory for G=10 or G=15. However, by experimenting with N=8, we can reduce the execution time even further without significantly compromising the algorithm's performance.

```
5 8 6 1 0.3 : [1.67, 9.84, 0.49, 0.99] in 3.43 sec
10 8 6 1 0.3 : [1.5, 9.75, 0.49, 2.66] in 7.11 sec
15 8 6 1 0.3 : [1.45, 9.69, 0.49, 4.05] in 8.46 sec
25 8 6 1 0.3 : [1.43, 9.66, 0.49, 5.93] in 13.51 sec
50 8 6 1 0.3 : [1.43, 9.65, 0.49, 7.04] in 25.58 sec
```

Thus, we conclude that a parameter configuration similar to the one mentioned above yields, on average, the best possible results. Let's revisit example1() with these parameters. The algorithm does not achieve the solution with the same precision as in example 1 due to L=6, but based on the above analysis, it is more likely to reach a minimum point compared to other parameter choices.

```
G = 12  #total generations
N = 8  #number of chromosomes per generation. Even number
L = 6  #binary length
C_MINMAX_FIT_PERCENTAGE = 0.01
Cross_prob = 1  #probability of crossover
Mutation_prob = 0.3  #probability of mutation
```

```
--Generation: 0--
Average Fitness:  6.771784411619627
Diversity:  0.49999999999999994
Best Solution:  2.0168951761646037
--Generation: 1--
Average Fitness:  6.489450167488089
Diversity:  0.38690476190476186
Best Solution:  2.0168951761646037
--Generation: 2--
Average Fitness:  11.525993484006394
Diversity:  0.5297619047619048
Best Solution:  4.199897777251248
--Generation: 3--
Average Fitness:  10.856495974689423
Diversity:  0.5416666666666666
Best Solution:  3.478563467967459
--Generation: 4--
Average Fitness:  13.864051288518418
Diversity:  0.42857142857142855
Best Solution:  2.582774400513264
--Generation: 5--
Average Fitness:  12.468542359059333
Diversity:  0.47023809523809523
Best Solution:  1.4786285289409156
--Generation: 6--
Average Fitness:  18.08041780847181
Diversity:  0.41666666666666663
Best Solution:  3.766201995991387
--Generation: 7--
Average Fitness:  10.467267595271327
Diversity:  0.5297619047619049
Best Solution:  2.6844426319246306
--Generation: 8--
Average Fitness:  13.188192013312436
Diversity:  0.38690476190476186
Best Solution:  3.107151450673723
--Generation: 9--
Average Fitness:  4.877206825483834
Diversity:  0.5595238095238095
Best Solution:  1.4286792893031288
--Generation: 10--
Average Fitness:  12.461917560384734
Diversity:  0.5416666666666667
Best Solution:  2.9746714928831874
--Last Generation: 11--
['011101', '100010', '111101', '011001', '001110', '000100', '001000', '010011']
Average Fitness:  11.227880420395493
Diversity:  0.5357142857142857
Best Solution:  1.4286792893031288
```

Let us note here that the algorithm generally exhibits high diversity (~0.5) and average fitness in each generation. This is due to the high mutation probability, which "shuffles" the population in every generation. This specific problem seems to favor GAs that operate with a high degree of randomness.

The conclusions drawn above do not consider potential interactions between the parameters. For example, the effect of Cross_prob on the algorithm might vary for different L values. The function example3() provides the same results as example2() but covers all combinations of parameters (a total of $5^5 = 3725$). By analyzing the results of example3(), one may identify parameter combinations that yield similar or better results. It is also worth mentioning that a GA can generally involve additional parameters, such as the use of different chromosome selection methods (rather than roulette selection—some of which will be discussed in Chapter 6), the use of Gray encoding (which will be covered on the next chapter), `C_MINMAX_FIT_PERCENTAGE = 0.01` which we ignored as a constant in order to focus on the main parameters but may provide improvements to the algorithm with different values, and more. The optimization of these parameters is not as straightforward as the simple case we presented.

Chapter 3 Gray Encoding

Having presented Genetic Algorithms (GAs) in their basic form and analyzed their performance, we will now examine two variations of the basic algorithm. The differences lie in the encoding of solutions, with each attempting to address a particular issue of the standard algorithm. One variation modifies the binary representation, while the other directly employs floating-point numbers without any intermediate encoding. In this chapter, we will focus on the first variation, while the two following chapters will cover the second one in more detail.

A key issue with standard binary encoding is its variability in representation [3]. For instance, altering a single bit in the chromosome 01001000=72 can result in 00001000=8, whereas changing 15=01111 by just 1 requires modifying all 5 bits of the chromosome: 16=10000. Generally, modifying the leftmost bits of a chromosome has a larger impact on the value. This problem can be avoided with **Gray Encoding**: in this method, consecutive numbers $m, m+1$ with binary representations δ_m, δ_{m+1} have a Hamming distance $d_H(\delta_m, \delta_{m+1}) = 1$. Table 3.1 presents the Gray representations for the natural numbers up to 15. Within the GA, the crossover and mutation processes take place in Gray encoding, with conversions performed to retrieve the actual values. The use of Gray encoding adds to the execution time by the amount of time required for these conversions, and in some applications, the computational cost may not be worth it. Program 2 provides functions for converting between the two encodings.

The rule for conversion from binary encoding to Gray encoding [12] is as follows: If $B_1B_2B_3B_4$... the binary encoding bits and $G_1G_2G_3G_4$... the Gray encoding bits:

a) The most important bit (B_1) of the binary representation remains the same in the Gray code, thus $G_1=B_1$.

b) The rest of the Gray code bits result from the pairwise XOR operation between the binary encoded bits, for example $G_2=B_1$ XOR B_2, $G_3=B_2$ XOR B_3, etc.

Conversely, for the conversion from Gray to binary representation, the following holds again:

a) $B_1=G_1$, while

b) the i-bit of the binary encoding results from the XOR operation between the (i-1)-bit of the binary encoding and the i-bit of the Gray encoding, for example $B_2=B_1$ XOR G_2, $B_3=B_2$ XOR G_3, etc.

Based on the forementioned algorithm, Table 3.1 contains the conversions between Binary and Gray encoding for $l = 4$.

Value	Binary Encoding	Gray Encoding	Value	Binary Encoding	Gray Encoding
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Table 3.1: Gray Encoding for the first 16 natural numbers

By setting the global variable GrayEncoding=1, in program 1, all the functions we reviewed in previous chapters-example1(), example2(), and example3()-are executed using Gray encoding instead of binary. Running example1() provides the following indicative execution of the GA using Gray encoding, with the same parameters as in Example 1, yields:

```
--Generation: 0--
Average Fitness:  7.4767357177438605
Diversity:  0.5396825396825397
Best Solution:  1.9683779879574579
--Generation: 1--
Average Fitness:  6.398086170890469
Diversity:  0.4722222222222222
Best Solution:  1.393686421024432
--Generation: 2--
Average Fitness:  5.5178778067544965
Diversity:  0.4007936507936507
Best Solution:  1.393686421024432
--Generation: 3--
Average Fitness:  7.430444597668817
Diversity:  0.4007936507936508
Best Solution:  5.005289724134325
--Generation: 4--
Average Fitness:  7.423754078791358
Diversity:  0.4920634920634921
Best Solution:  5.880126504164607
--Generation: 5--
Average Fitness:  7.2381085432044125
Diversity:  0.5555555555555555
Best Solution:  5.880126504164607
--Generation: 6--
Average Fitness:  7.264684936385864
Diversity:  0.44841269841269843
Best Solution:  5.040951353089425
--Generation: 7--
Average Fitness:  6.520183670176497
Diversity:  0.4126984126984127
Best Solution:  2.0533562387007525
--Generation: 8--
Average Fitness:  6.440037106004532
Diversity:  0.29761904761904756
Best Solution:  5.880126504164607
--Last Generation: 9--
```

```
['101101000', '101101000', '010001000', '101101001', '101101000', '101101000', '110010111',
'101010111']
Average Fitness:  6.410109717987686
Diversity:  0.4325396825396825
Best Solution:  3.3277548118829947
```

The population of the last generation is represented in binary encoding, as in this implementation Gray encoding is only used for the crossover and mutation stages. The algorithm identifies the global minimum on the first few generations but fails to preserve it for more generations. Let's analyze the overall GA performance, with the use of Gray encoding, via the function example2(), as in Chapter 2. Table 3.2 corresponds to Table 2.

5 8 9 0.8 0.01 : [2.37, 9.54, 0.47, 0.6] in 2.4 sec
10 8 9 0.8 0.01 : [2.2, 9.04, 0.45, 1.6] in 6.41 sec
15 8 9 0.8 0.01 : [2.12, 8.75, 0.44, 2.59] in 9.73 sec
25 8 9 0.8 0.01 : [2.02, 8.3, 0.42, 5.17] in 12.4 sec
50 8 9 0.8 0.01 : [1.87, 7.71, 0.4, 13.05] in 27.04 sec
10 4 9 0.8 0.01 : [3.53, 8.69, 0.45, 1.39] in 3.48 sec
10 8 9 0.8 0.01 : [2.2, 9.04, 0.45, 1.6] in 5.81 sec
10 12 9 0.8 0.01 : [1.87, 9.49, 0.47, 1.76] in 9.28 sec
10 16 9 0.8 0.01 : [1.69, 9.53, 0.48, 2.01] in 14.67 sec
10 20 9 0.8 0.01 : [1.58, 9.82, 0.48, 2.07] in 23.55 sec
10 8 5 0.8 0.01 : [2.25, 10.16, 0.47, 0.89] in 5.22 sec
10 8 6 0.8 0.01 : [2.2, 9.78, 0.47, 1.04] in 5.88 sec
10 8 9 0.8 0.01 : [2.2, 9.04, 0.45, 1.6] in 5.97 sec
10 8 12 0.8 0.01 : [2.25, 8.96, 0.46, 1.6] in 5.93 sec
10 8 15 0.8 0.01 : [2.31, 8.88, 0.45, 1.69] in 5.97 sec
10 8 20 0.8 0.01 : [2.34, 8.95, 0.46, 1.88] in 6.27 sec
10 8 9 1 0.01 : [2.18, 8.97, 0.46, 1.56] in 7.16 sec
10 8 9 0.8 0.01 : [2.2, 9.04, 0.45, 1.6] in 5.91 sec
10 8 9 0.7 0.01 : [2.22, 8.91, 0.45, 1.66] in 5.37 sec
10 8 9 0.5 0.01 : [2.25, 9.04, 0.45, 1.58] in 5.31 sec
10 8 9 0.3 0.01 : [2.36, 9.03, 0.44, 1.6] in 5.58 sec
10 8 9 0.8 0 : [2.44, 9.55, 0.47, 0.76] in 4.75 sec
10 8 9 0.8 0.01 : [2.2, 9.04, 0.45, 1.6] in 4.97 sec
10 8 9 0.8 0.05 : [1.82, 8.8, 0.47, 2.83] in 6.15 sec
10 8 9 0.8 0.1 : [1.66, 8.98, 0.48, 3.1] in 5.8 sec
10 8 9 0.8 0.3 : [1.48, 9.44, 0.5, 3.4] in 5.72 sec

Table 3.2

Comparing Table 3.2 with Table 2.1, we observe that the differences in performance metrics for minimizing this specific fitness function are almost negligible. Therefore, we can assume that Gray encoding does not lead to any significant improvement in the algorithm's performance. The parameter combination we previously identified as optimal in Chapter 2 (N=8, L=6, Cross_prob=1, Mutation_prob=0.3) is likely to yield similarly satisfactory results with Gray encoding as well. According to the literature (see, e.g., [2]), Gray encoding performs better than binary encoding for some problems, while for others, like in this example, it offers no substantial advantage.

Chapter 4 RGA

The use of binary encoding, whether Gray or standard, can be avoided by employing Real-Valued Genetic Algorithms (RGAs). It's important to clarify that this approach isn't necessarily an improvement over previous encoding methods but rather a variation. Let's examine how an RGA operates in a manner analogous to the processes of the basic algorithm.

Consider an optimization problem with K parameters. We represent chromosomes as lists of parameters p_i: $[p_1, \dots, p_i, \dots, p_K]$. The parameters p_i are represented as floating-point numbers normalized within the range $(0,1)$. For their use in the algorithm, they are linearly denormalized using a function similar to the "Remap" function from the basic algorithm:

$$Remap(x) = (b - a)x + a.$$

This way, there is no need for extra bits to enhance the algorithm's precision; we achieve the maximum precision the computer can provide right from the start. The initial population consists of N such lists. Fitness calculation and chromosome selection for crossover are performed as in the basic algorithm.

A first approach for crossover, analogous to the basic algorithm, would involve splitting the list into two parts and swapping them in a crossover fashion. For example:

$$[p_{11}, p_{12}|p_{13}] \rightarrow [p_{11}, p_{12}|p_{23}]$$
$$[p_{21}, p_{22}|p_{23}] \rightarrow [p_{21}, p_{22}|p_{13}]$$

Other splitting methods, which we will discuss later, can also be applied. In any case, we haven't gained much since all p_{ij} values already existed; we haven't introduced new information for the algorithm's progression. One such method relies entirely on mutation and may only yield satisfactory results in certain problems with appropriate mutation parameters.

We will delve into mutation in RGAs after proposing an alternative to the crossover mentioned earlier: blending methods. These methods create new parameter values by combining those of the parents. The simplest way is through linear combination, for instance, for the second parameter in the example above:

$$p_{new2} = \beta p_{12} + (1 - \beta)p_{22}, \qquad \beta \in [0,1]$$

If $\beta = 0$ or $\beta = 1$ one of the two parameters dominates the other, while for $\beta = 0.5$ we take the average of the two parameters. Coefficient β, usually random in range $[0,1]$, can either be the same or different for each parameter in the new chromosome. Moreover, its selection doesn't have to come from a uniform distribution and can even allow for non-convex combinations of parameters, meaning $\beta > 1$ or $\beta < 0$, always within the problem's constraints. If a combination exceeds these limits, re-selection occurs. Blending can happen for one or more parameters and can be combined with the splitting method described above. For example, a possible version of the algorithm might involve blending for the second parameter, splitting and crossing over for the third and rightmost parameter, giving us:

$$[p_{11}, p_{12}|p_{13}, p_{14}] \rightarrow [p_{11}, \beta p_{12} + (1-\beta)p_{22}|p_{23}, p_{24}]$$
$$[p_{21}, p_{22}|p_{23}, p_{24}] \rightarrow [p_{21}, \beta' p_{12} + (1-\beta')p_{22}|p_{13}, p_{14}]$$

For the mutation process, we can no longer simply "flip" 0 to 1 and vice versa. Instead, with a probability p_μ, we can add some white noise:

$$p_{\text{mutated}} = p + \sigma * N(0,1)$$

Where σ is the standard deviation of the normal distribution. The choice of σ can be arbitrary or depend on the generation or the size of the parameter's domain. Alternatively, the mutated parameter may take on a completely random new value within its domain. Let's look at an example from an RGA algorithm with $\sigma = 1.5$, for finding the minimum of the function in Figure 1.3. Suppose that in some generation, the following chromosomes are selected for crossover as part of a population. Let's see what follows in the table below:

Chromosome/ Parameter x	Is Crossover happening?	Random $\beta \in [0,1]$	x_{new}
4.52	Yes	0.24	$0.24 * 4.52 + (1-0.24) * (-3.45) = -1.54$
-3.45		0.57	$0.57 * 4.52 + (1-0.57) * (-3.45) = 1.09$
-7.2	Yes	0.93	$0.93 * (-7.2) + (1-0.93) * (1.39) = -6.6$
1.39		0.17	$0.17 * (-7.2) + (1-0.17) * (1.39) = -0.07$

x_{new}	Is Mutation happening?	Random $N(0,1)$	$x_{mutated}$ $\sigma = 1.5$, σταθερό	$x_{next\ gen}$
-1.54	Yes	0.55	$-1.54 + 1.5 * 0.55 = -0.71$	-0.71
1.09	No	-	1.09	1.09
-6.6	No	-	-6.6	-6.6
-0.07	Yes	-0.45	$-0.07 - 1.5 * 0.45 = -0.74$	-0.74

Table 4.1

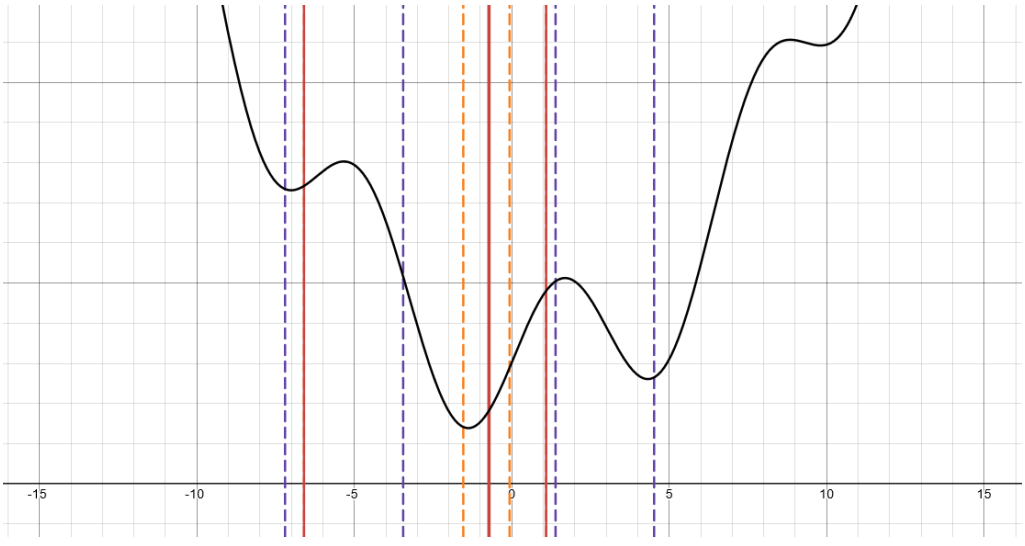

Figure 4.1: Purple dashed lines representing the initial x, orange dashed lines for the x_{new}, and red for the $x_{next\ gen}$ (as expected, there is overlap between the red and orange lines at 1.09 and -6.6, while two red lines end up very close to each other)

Let's now revisit the problem from the previous chapters using an RGA. Program 3 is a variation of Program 1, designed to function as an RGA. Many functions in the two programs are shared, and the two could easily be combined into a single unified program. However, for clarity in presentation, we will keep them as two separate programs. Running the example1() function, with parameters:

```
G = 10  #total generations
N = 8  #number of chromosomes per generation. Even number
sigma = 1
C_MINMAX_FIT_PERCENTAGE = 0.01
Cross_prob = 0.8  #probability of crossover
Mutation_prob = 0.2 #probability of mutation
a = -15  #domain [a,b]
b = 15
rd.seed(3)
```

Meaning an RGA with 10 generations, 8 candidate solutions per generation, 0.8 crossover probability, 0.01 mutation probability, we obtain the following indicative execution of the Genetic Algorithm:

```
--Generation: 0--
Average Fitness:  10.170134979343729
Best Solution:  2.8920264939395897
--Generation: 1--
Average Fitness:  4.874125364323016
Best Solution:  2.5544032065877067
--Generation: 2--
Average Fitness:  5.233670001012929
Best Solution:  2.5544032065877067
--Generation: 3--
Average Fitness:  4.1356096001746625
Best Solution:  2.2396159812047562
--Generation: 4--
Average Fitness:  3.9115890961375075
Best Solution:  3.048302492395437
--Generation: 5--
Average Fitness:  3.1961405226272186
Best Solution:  3.048302492395437
--Generation: 6--
Average Fitness:  3.1762883768940173
Best Solution:  2.7467047434678533
--Generation: 7--
Average Fitness:  3.013068768176997
Best Solution:  1.4522654280697864
--Generation: 8--
Average Fitness:  2.7583510426076803
Best Solution:  1.4522654280697864
--Last Generation: 9--
[-0.12062796989772513, 0.047491112502553584, -0.4842661132458064, 0.047491112502553584,
-0.9639376723365033, -1.999894812333376, 0.047491112502553584, 0.047491112502553584]
Average Fitness:  2.600373467475479
Best Solution:  1.5714242601160922
```

By repeatedly running example2() for all parameter combinations, as in Table 2.1, we obtain Table 4.2. On the left, we see the set of parameters in order: G (number of generations), N (population size), sigma, Cross_prob (crossover probability), and Mutation_Prob (mutation probability). On the right, we see the following averaged results, based on multiple runs (reps=100): the minimum fitness (min_fit) achieved by the algorithm in any of its generations, the average fitness of the population (avg_fit) during the generation where the min_fit is achieved, and the average generation at which this minimum fitness is reached.

5 8 1 0.8 0.2 : [1.93, 7.2, 1.06] in 0.17 sec
10 8 1 0.8 0.2 : [1.88, 6.2, 2.43] in 0.35 sec
15 8 1 0.8 0.2 : [1.86, 5.75, 4.01] in 0.55 sec
25 8 1 0.8 0.2 : [1.79, 5.14, 8.92] in 0.73 sec
50 8 1 0.8 0.2 : [1.75, 4.67, 18.65] in 1.34 sec
10 4 1 0.8 0.2 : [3.4, 8.2, 1.23] in 0.14 sec
10 8 1 0.8 0.2 : [1.89, 6.53, 2.19] in 0.37 sec
10 12 1 0.8 0.2 : [1.51, 5.41, 2.86] in 0.73 sec
10 16 1 0.8 0.2 : [1.55, 5.44, 3.21] in 1.1 sec
10 20 1 0.8 0.2 : [1.44, 4.93, 3.85] in 1.6 sec
10 8 0.1 0.8 0.2 : [1.96, 7.05, 1.58] in 0.36 sec
10 8 0.5 0.8 0.2 : [1.93, 6.38, 2.19] in 0.4 sec
10 8 1 0.8 0.2 : [1.88, 6.2, 2.43] in 0.4 sec
10 8 2 0.8 0.2 : [1.78, 5.89, 2.92] in 0.38 sec
10 8 3 0.8 0.2 : [1.72, 5.93, 3.0] in 0.36 sec
10 8 1 1 0.2 : [1.76, 5.78, 2.35] in 0.35 sec
10 8 1 0.8 0.2 : [1.89, 6.53, 2.19] in 0.35 sec
10 8 1 0.7 0.2 : [1.92, 6.54, 2.41] in 0.3 sec
10 8 1 0.5 0.2 : [1.99, 6.73, 2.58] in 0.31 sec
10 8 1 0.3 0.2 : [2.08, 7.29, 2.33] in 0.36 sec
10 8 1 0.8 0 : [1.98, 7.05, 1.47] in 0.68 sec
10 8 1 0.8 0.1 : [1.88, 6.52, 2.1] in 0.49 sec
10 8 1 0.8 0.2 : [1.93, 6.43, 2.3] in 0.45 sec
10 8 1 0.8 0.3 : [1.87, 6.06, 2.43] in 0.45 sec
10 8 1 0.8 0.5 : [1.74, 5.56, 3.25] in 0.55 sec

Table 4.2

Based on the table we can derive similar conclusions to the ones from Chapter 2. To summarize:

- Larger G allows for better solutions at the cost of more computations
- Larger N allows for better solutions at the cost of more computations
- Higher crossover probability gives slightly better results
- Higher mutation probability gives better results

Additionally, we observe that a larger standard deviation (sigma) favors finding better solutions after slightly more generations in average. Based on these observations, we can choose a set of parameters that will likely be optimal for this specific problem. We select N=8, sigma=3, Cross_prob=1, Mutation_prob=0.5 and run the function example2() (reps=1000) for these parameters, looping over G_list.

```
5 8 3 1 0.5 : [1.7, 6.63, 1.25] in 4.85 sec
10 8 3 1 0.5 : [1.61, 5.59, 3.4] in 6.48 sec
15 8 3 1 0.5 : [1.57, 5.08, 5.68] in 8.86 sec
25 8 3 1 0.5 : [1.5, 4.46, 10.84] in 10.98 sec
50 8 3 1 0.5 : [1.43, 3.7, 24.5] in 28.91 sec
```

Comparing these results to the ones from chapter 2, using the optimal parameters we had identified:

```
5 8 6 1 0.3 : [1.67, 9.84, 0.49, 0.99] in 3.43 sec
10 8 6 1 0.3 : [1.5, 9.75, 0.49, 2.66] in 7.11 sec
15 8 6 1 0.3 : [1.45, 9.69, 0.49, 4.05] in 8.46 sec
25 8 6 1 0.3 : [1.43, 9.66, 0.49, 5.93] in 13.51 sec
50 8 6 1 0.3 : [1.43, 9.65, 0.49, 7.04] in 25.58 sec
```

We observe that the RGA achieves approximately the same average optimal performance, but typically requires more generations and slightly more time. Additionally, we notice that when the RGA finds its optimal value, the population exhibits a lower average fitness, indicating that the algorithm is converging towards the solution. In the literature, there are proponents of both binary-coded GAs and RGAs. [7]

Chapter 5 Application and Visualization

In this chapter, we will present an application of a RGA with two parameters (x, z) for maxima identification. Specifically, we will explore four examples of objective functions. The goal of the application is to explain and understand how the parameters of an RGA affect its operation and performance. We achieve this by visualizing the algorithm in the Houdini FX graphics environment. This software was chosen due to its low-level access to geometric element data, its scripting capabilities, its speed and its visualization options. By visualizing the objective function and the algorithm's candidate solutions, we get a clear picture of the algorithm's behavior, especially since we already know certain characteristics of the objective function, such as the number, density, and "elevation differences" of local extrema. Additionally, we will observe that different parameter choices in the algorithm favor different types of objective functions. Let's once again examine the algorithm's parameters as they appear in the application interface (Image 5.1).

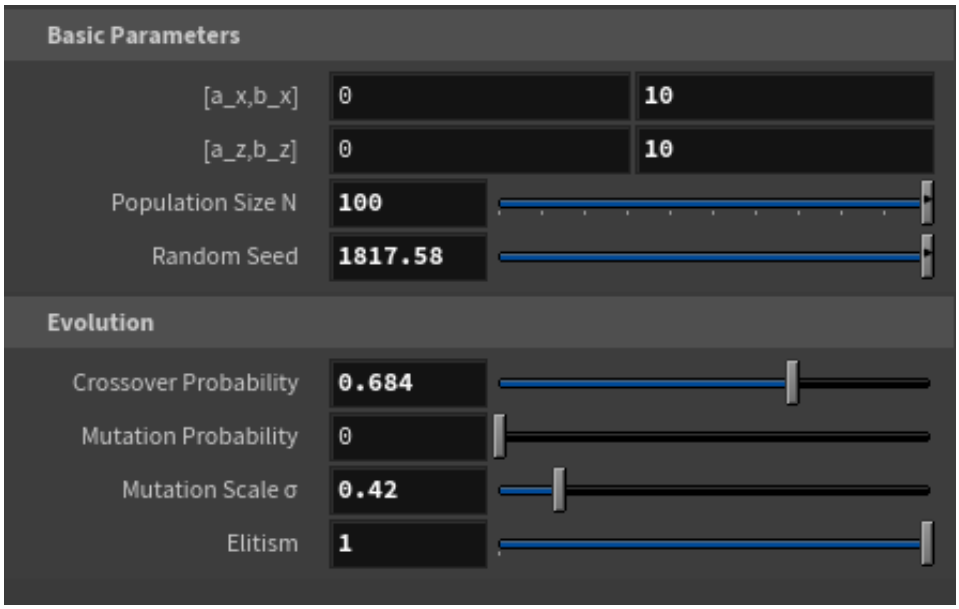

Image 5.1: Application parameter settings

- Domain: [a_x, b_x]x[a_z, b_z]
- Population Size N: The number of individuals in the population (denoted as pop)
- Random Seed: The initialization of the pseudo-random number generator (denoted as seed).
- Crossover Probability (denoted as pc)
- Mutation Probability (denoted as pm)
- Mutation Scale σ: The standard deviation for the mutation process (denoted as sigma or σ)
- Elitism: A binary parameter (1 or 0), determining whether elitism is active in the selection process.

Most of the parameters have been extensively analyzed in previous chapters. Different initializations of the pseudo-random number generator change all random processes within the

algorithm. In practice, this means that each different Random Seed corresponds to a different experiment/trial of the stochastic process of the GA. A similar concept appears in programs 1 and 3. The Elitism parameter has two options, 0 or 1, to activate or deactivate it. Elitism in GAs refers to favoring the best solutions in the population. In this simpler version of elitism used in our application, its activation ensures that the individual with the highest fitness is preserved into the next generation with a probability of 1. This means that the best solution (so far) is retained across generations, preserving the optimal information discovered by the algorithm.

We will first explain some general application elements and then execute the algorithm. Figures 5.2 - 5.6 show parts of the UI with explanations of their functions.

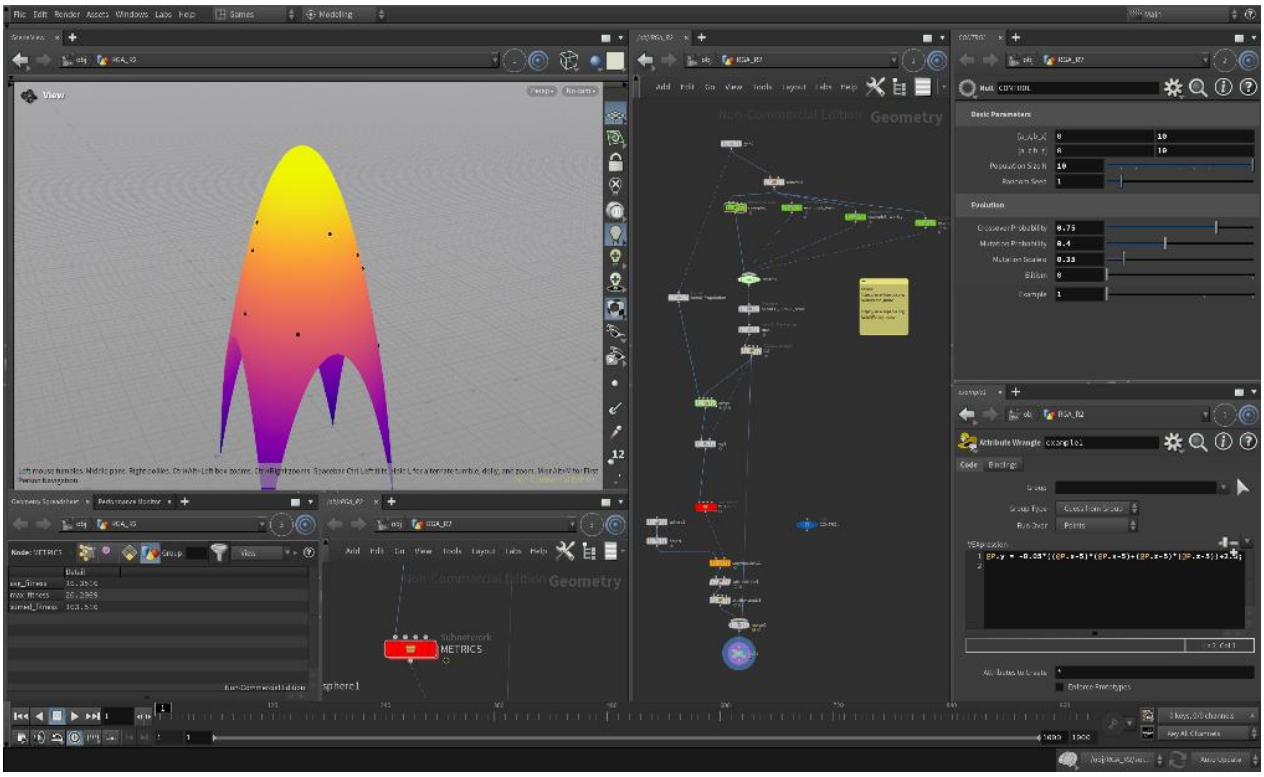

Image 5.2: Application UI

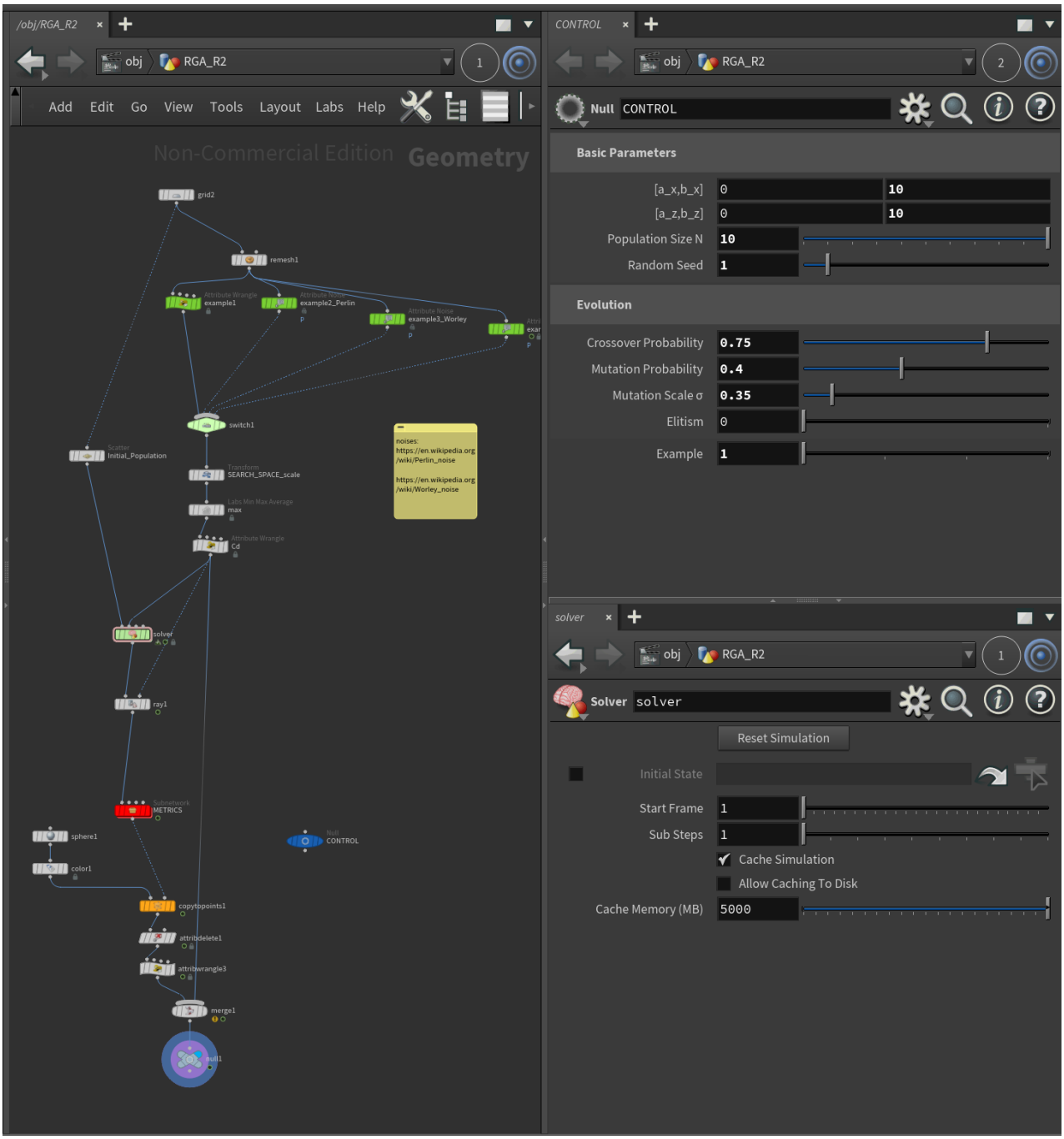

Image 5.3: On the left side, the node tree that forms the application. Inside the solver node lies the algorithm's core, as shown in Image 5.4. At the top right are the algorithm's parameters, while at the bottom right, we find the properties of the selected node from the node tree

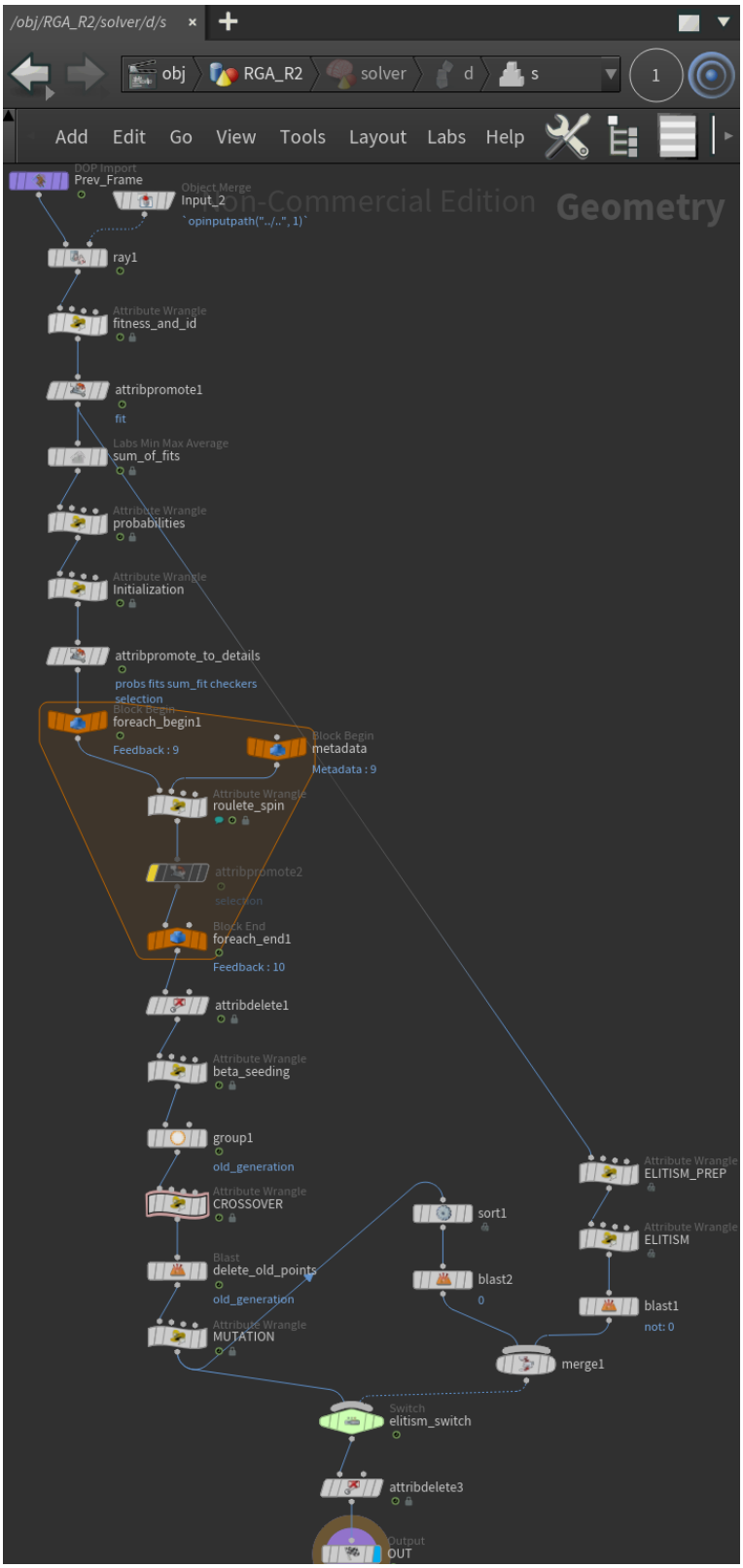

Image 5.4: The solver node contains a node tree that takes the previous generation as input and returns the current generation. It performs the selection, crossover, and mutation stages based on the given parameters

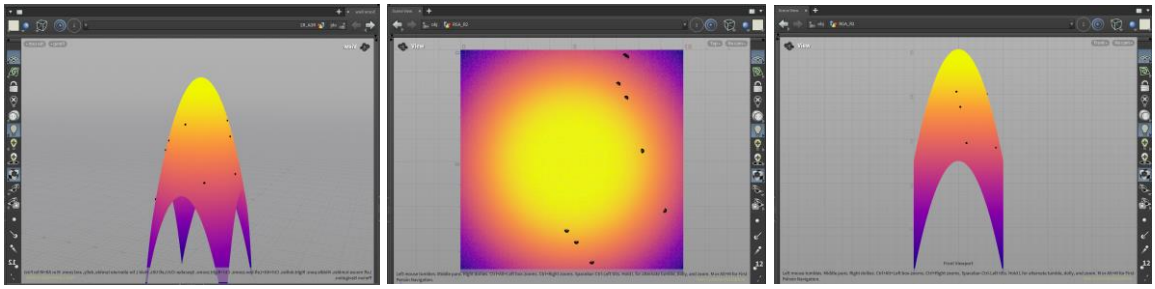

Image 5.5: On the left perspective 3D view, on the right oblique view and in the center, a top-down view of the objective function with color-coded height of the function.

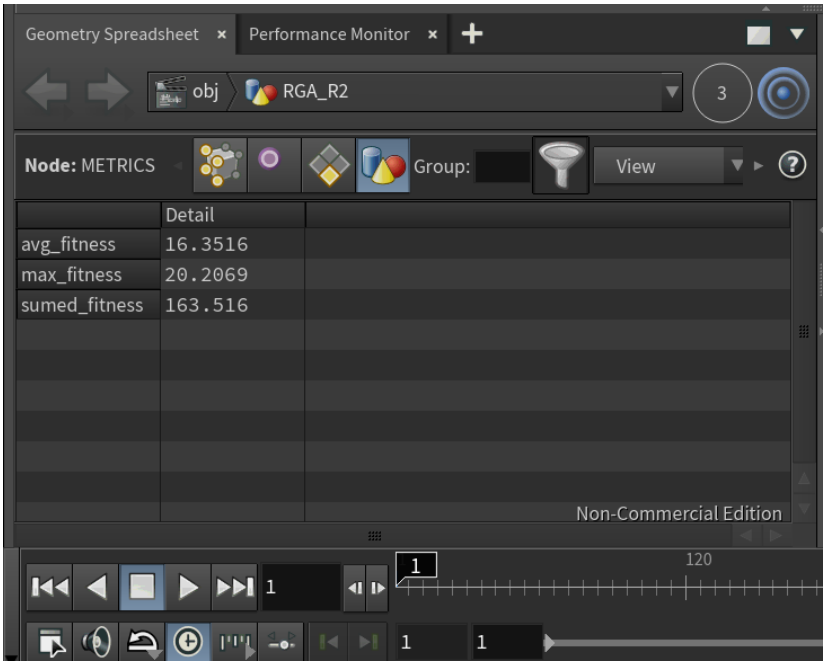

Image 5.6: Average and maximum fitness of the population at the current generation. On the timeline below we see the current generation number, here generation 1 (In this application Initial Population is generation 0)

In Image 5.5, the black dots on the objective function represent the members of the population. The solutions are located on the xz-plane, as shown in Image 5.7. However, we display them on the surface for visualization purposes. The parameters we will use until we make any changes, are those shown in Image 5.3.

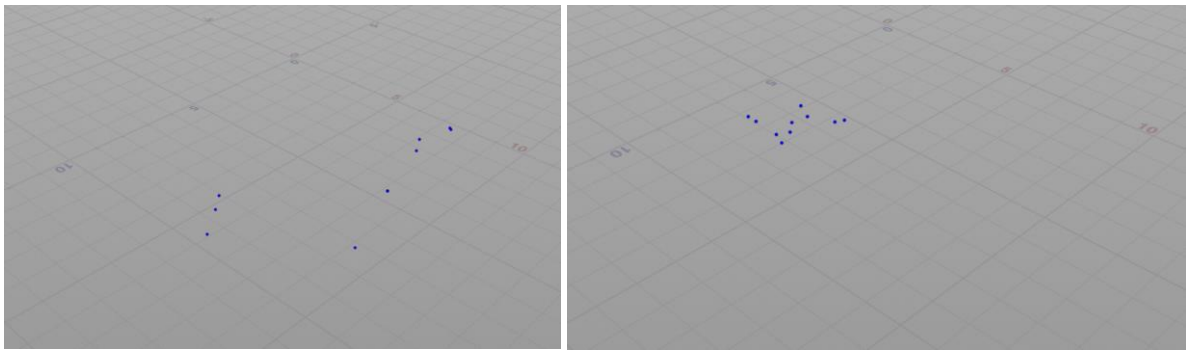

Image 5.7: Population of 10 candidate solutions without surface mapping, left seed=1, right seed=3.05

Let’s start with the first objective function example, the paraboloid presented in Image 5.5:

$$y = -0.5((x-5)^2 + (z-5)^2) + 25$$

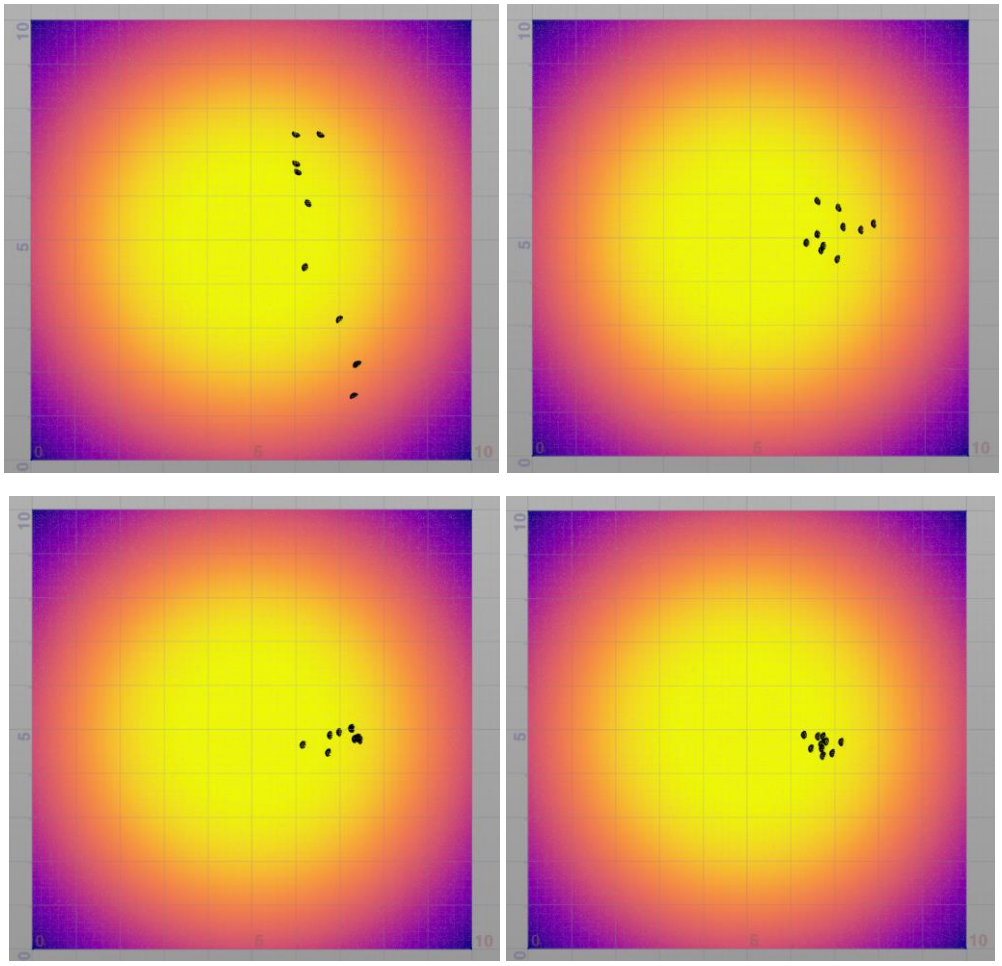

Image 5.8: Generations 3, 9, 22, and 42 are shown in the top left, top right, bottom left, and bottom right, respectively. Parameters: pop=10, seed=1, pc=0.75, pm=0.4, sigma=0.35, elitism=0

Running the algorithm, we observe that, over the course of generations, the algorithm's solutions seem to converge toward a point near the true global maximum (the point (5,5)), achieving an average fitness of 24.3773 and a maximum fitness of 24.1067. If the algorithm is allowed to continue running, the population will hover in a region close to the actual maximum. This behavior confirms that genetic algorithms (GAs) are considered search algorithms rather than classical optimizers. This phenomenon is not necessarily problematic, as is often sufficient to find a solution in the vicinity of the optimum. Starting from this point, one can apply a traditional optimization algorithm to pinpoint the extremum with the desired precision. Algorithms that combine genetic algorithms with other optimization techniques are called hybrids (see, e.g., [3]).

In this specific example, we can easily prevent the population from "wandering" by activating elitism, as shown in the diagrams of Image 5.9. We observe that the population retains the optimal solution, and after just a few generations, it mutates around the point (5,5), achieving an average

fitness of 24.9225 and a maximum fitness of 24.9989 within 25 generations (very close to the function's value of 25 at (5,5)).

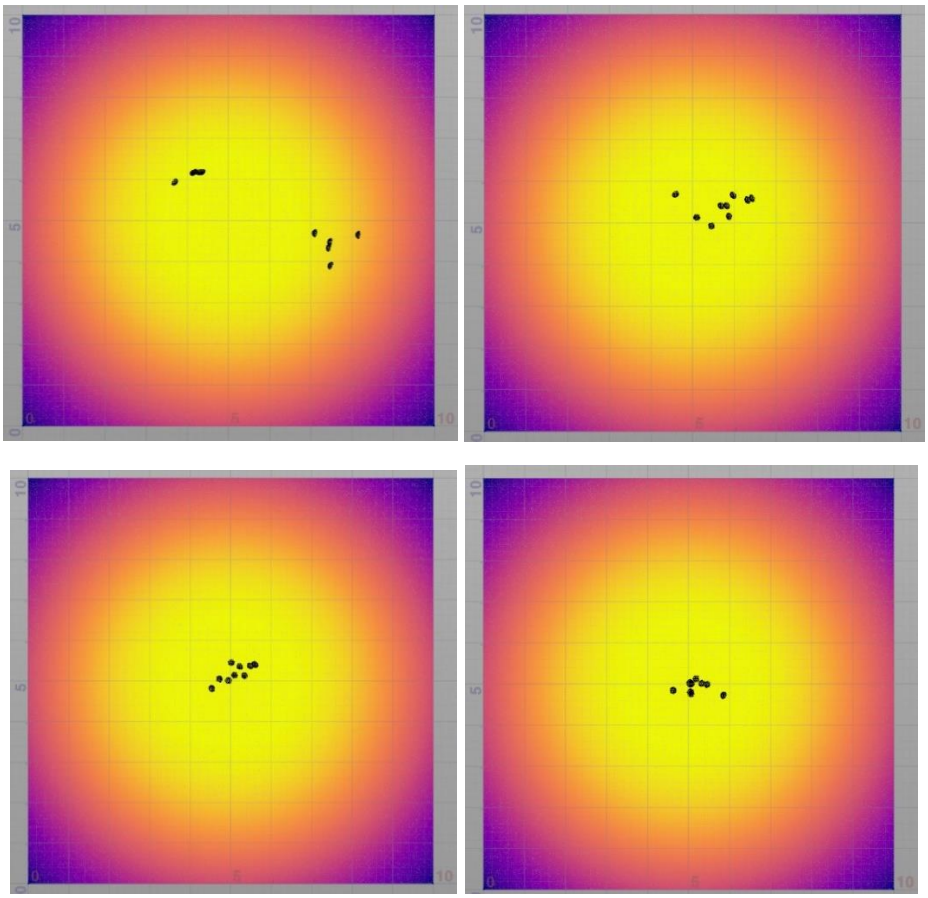

Image 5.9: Generations 5, 10, 15, 25 are shown in the top left, top right, bottom left, and bottom right, respectively. Parameters: pop=10, seed=1, pc=0.75, pm=0.4, sigma=0.35, elitism=1

This example represents a very simple case of an objective function, which is a quadratic form. As a result, finding the global optimum is relatively straightforward and can be achieved more efficiently, for example, using the Newton method, where the optimum is computed in a single step [9]. For the next two examples, we will use more complex surfaces as objective functions. Without delving into details from the field of computer graphics, these surfaces are generated from noises, algorithms that generate functions with desirable properties, such as continuity and certain orders of differentiability [16]. The objective function in the second example is derived from Perlin type noise, displayed in the diagrams of Image 5.10. In the application, for all the following examples, there is the option to modify the parameters of the noise to experiment with different objective functions. Similarly to the first example, one can also input any analytical function.

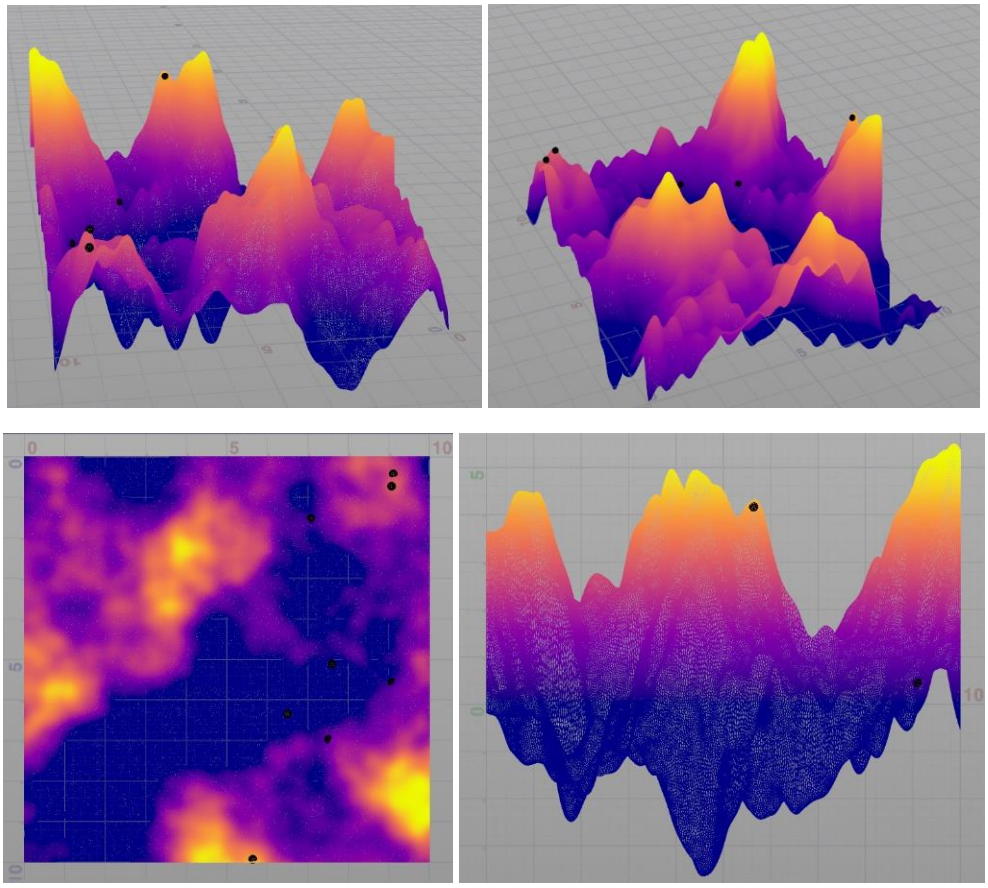

Image 5.10: Objective function, example 2

Let's run the genetic algorithm using the initial parameters.

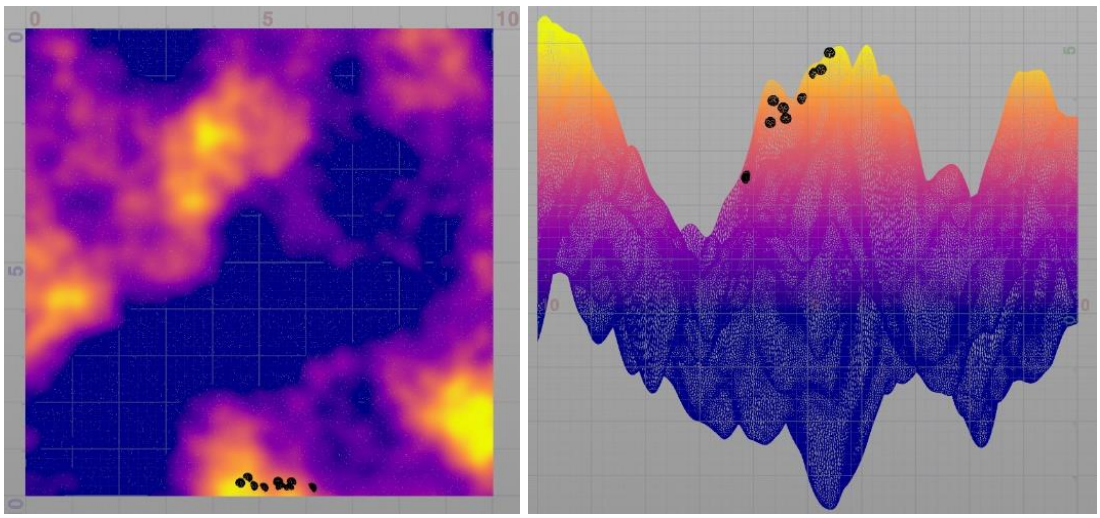

Image 5.11: By the 20th generation, the algorithm has located a local extremum, and the population becomes trapped around it. Parameters: pop=10, seed=1, pc=0.75, pm=0.4, sigma=0.35, elitism=0

By the 20th generation, the algorithm has identified a local maximum of the surface, as shown in Image 5.11. When elitism is enabled, we observe a similar behavior to the previous example: the

population no longer hovers around the local maximum and is able to identify the optimum with greater precision. In the 20th generation, without elitism, the algorithm achieves a maximum and average fitness of 4.7697 and 3.7139, respectively, compared to 4.9423 and 4.5672 with elitism enabled, see Image 5.12.

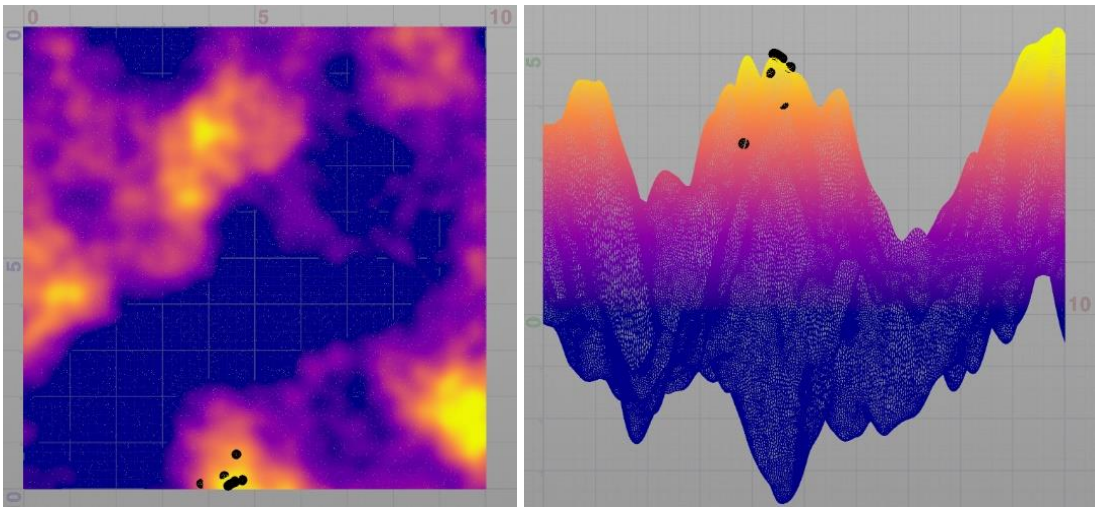

Image 5.12: 20th generation. Parameters: pop=10, seed=1, pc=0.75, pm=0.4, sigma=0.35, elitism=1

Now, let's try modifying some of the remaining parameters to observe how they affect the algorithm's behavior. By setting $\sigma = 0.1$, as shown in Image 5.13, we notice that the population gets stuck at a lower local maximum, close to the previous optimum. This occurs because a small value of σ restricts the mutated members of the population from moving far enough from their previous positions. Restoring σ to 0.35, we then increase the population size to 50. Starting with a larger initial population allows us to cover a larger portion of the search space, increasing the likelihood of finding better solutions. In Image 5.14, we observe that by the 12th generation, the algorithm has already located the global maximum. A larger population, combined with elitism, can lead to finding the global optimum even with smaller σ values, such as $\sigma = 0.15$, which we previously noted were less favorable for the algorithm's performance. In Image 5.15, we see the results over the generations. For around 150 generations, nearly the entire population is at the local maximum, with at least one optimal solution remaining at the global maximum due to elitism. After 150 generations, random mutations cause the entire population to "migrate" to the global maximum.

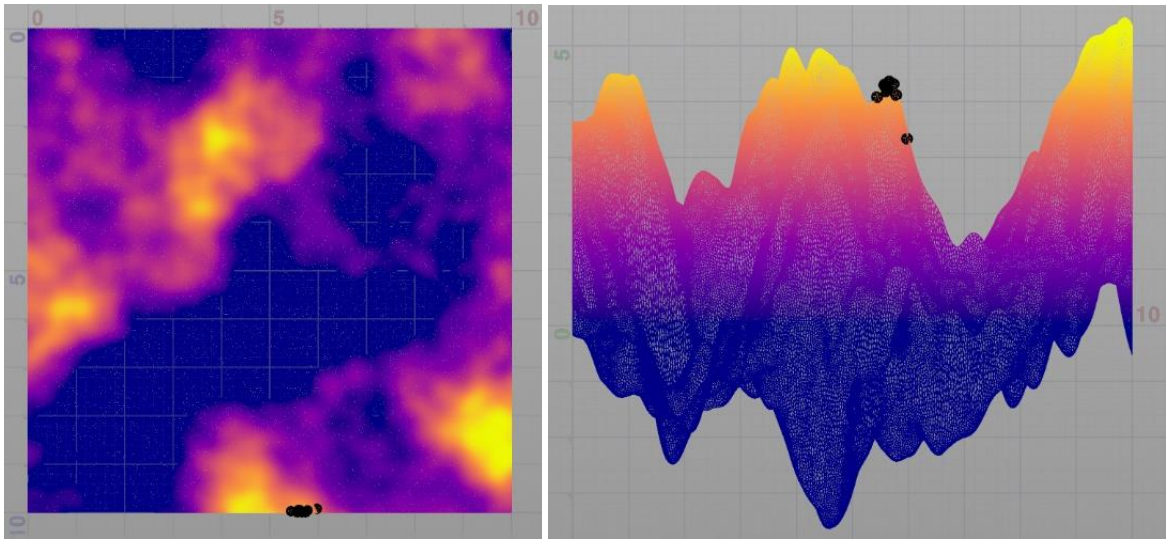

Image 5.13: 60th generation. Parameters: pop=10, seed=1, pc=0.75, pm=0.4, sigma=0.1, elitism=1

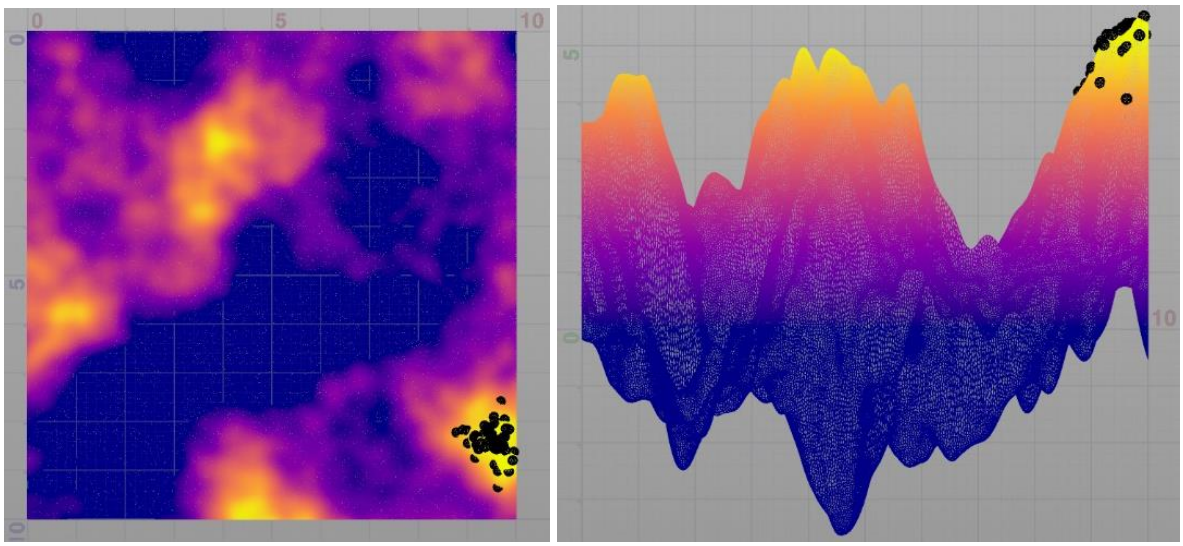

Image 5.14: 12th generation. Parameters: pop=50, seed=1, pc=0.75, pm=0.4, sigma=0.35, elitism=1

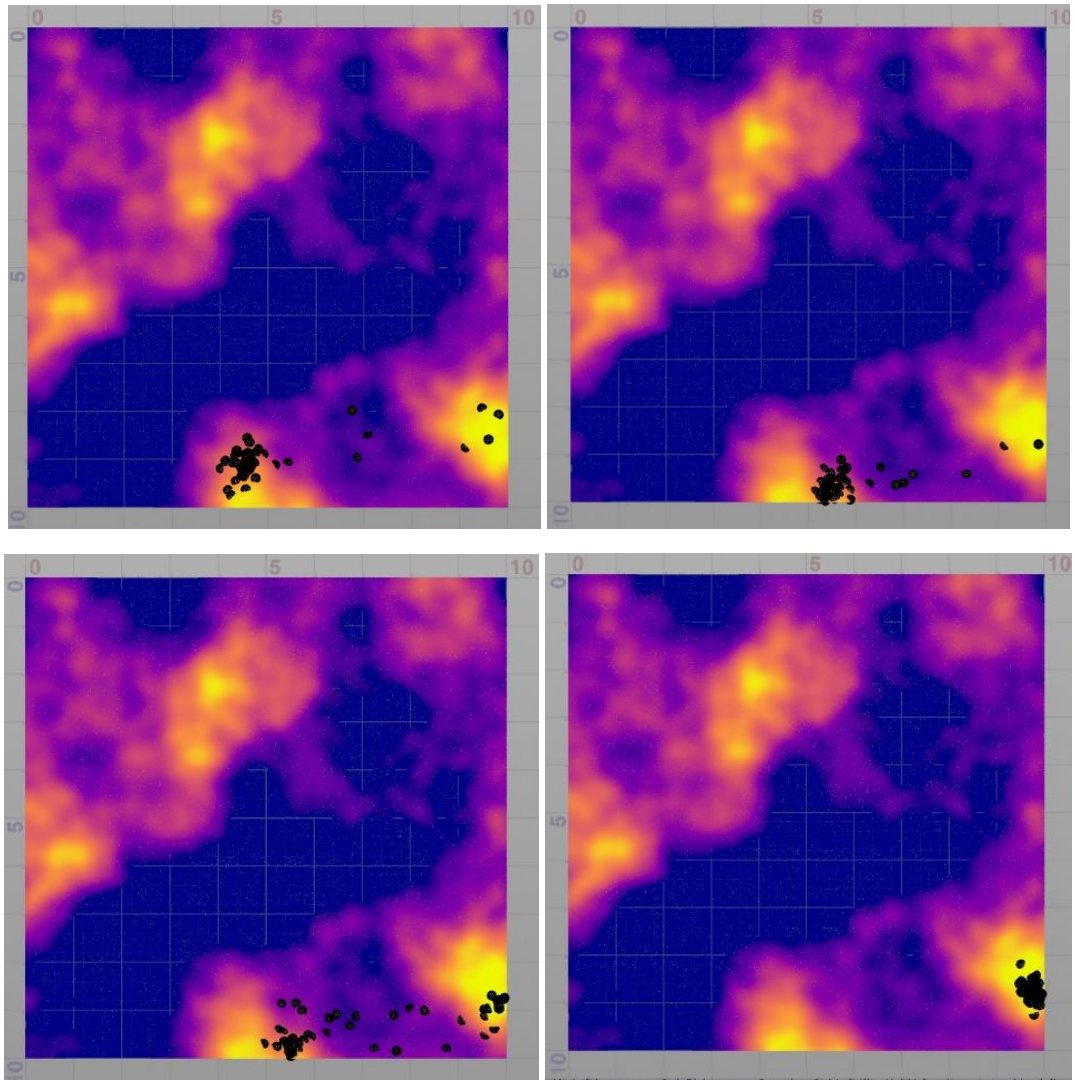

Image 5.15: Generations 10, 147, 151, 160 are shown in the top left, top right, bottom left, and bottom right, respectively. Parameters: pop=50, seed=1, pc=0.75, pm=0.4, sigma=0.15, elitism=1

However, we have already concluded from the previous chapters that increasing the population size leads to a corresponding increase in computational cost. Therefore, the goal is to adjust parameters that do not significantly affect the program's execution time. By identifying such "good" parameter combinations, we can consistently achieve solid results while reducing computational time. For example, we can try a population of 15 members with σ=2.05 (Image 5.16). Similar to Image 5.15, the population again "migrates" after several generations from the local to the global maximum. Here, however, the different parameters better facilitate this behavior. Specifically, the larger σ allows some members of the population to move from the local to the global optimum via mutation. It is worth mentioning that in the example of Image 5.15, unlike that of Image 5.16, the solution maintained at the global maximum by elitism existed from the early generations and was not the result of mutation. Specifically, without this solution, the algorithm would have been trapped at the local maximum.

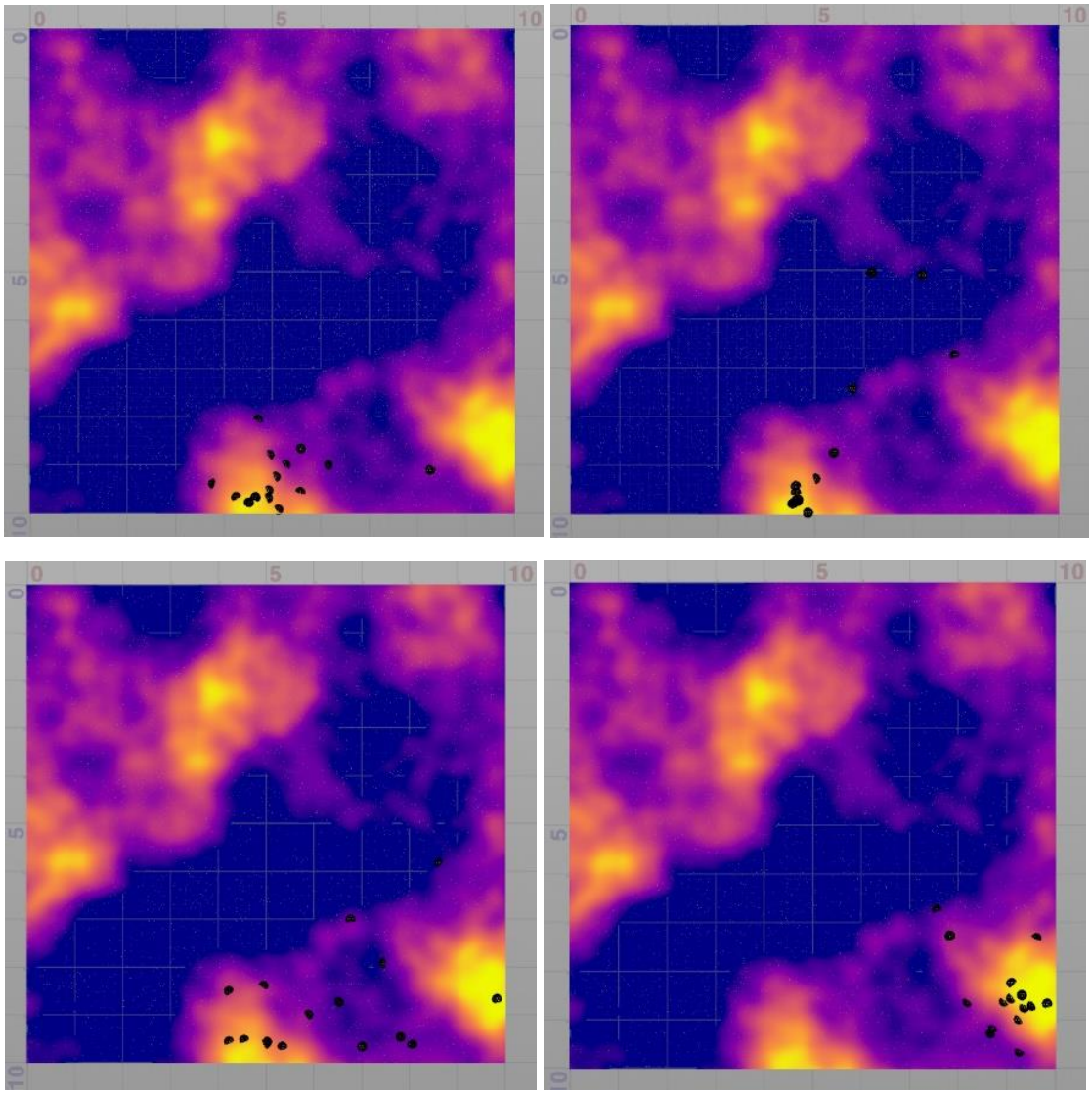

Image 5.16: Generations 10, 40, 57, 63 are shown in the top left, top right, bottom left, and bottom right, respectively. Parameters: pop=15, seed=1, pc=0.75, pm=0.4, sigma=2.05, elitism=1

For the third objective function example, after having explored the impact of elitism, population size, and σ, we will now focus on the parameters pc and pm. The objective function is generated using Worley noise [10] and is illustrated in Image 5.17. We will begin with a population of 100 members due to the density of local maxima, and we will deactivate elitism to observe the results without its interference.

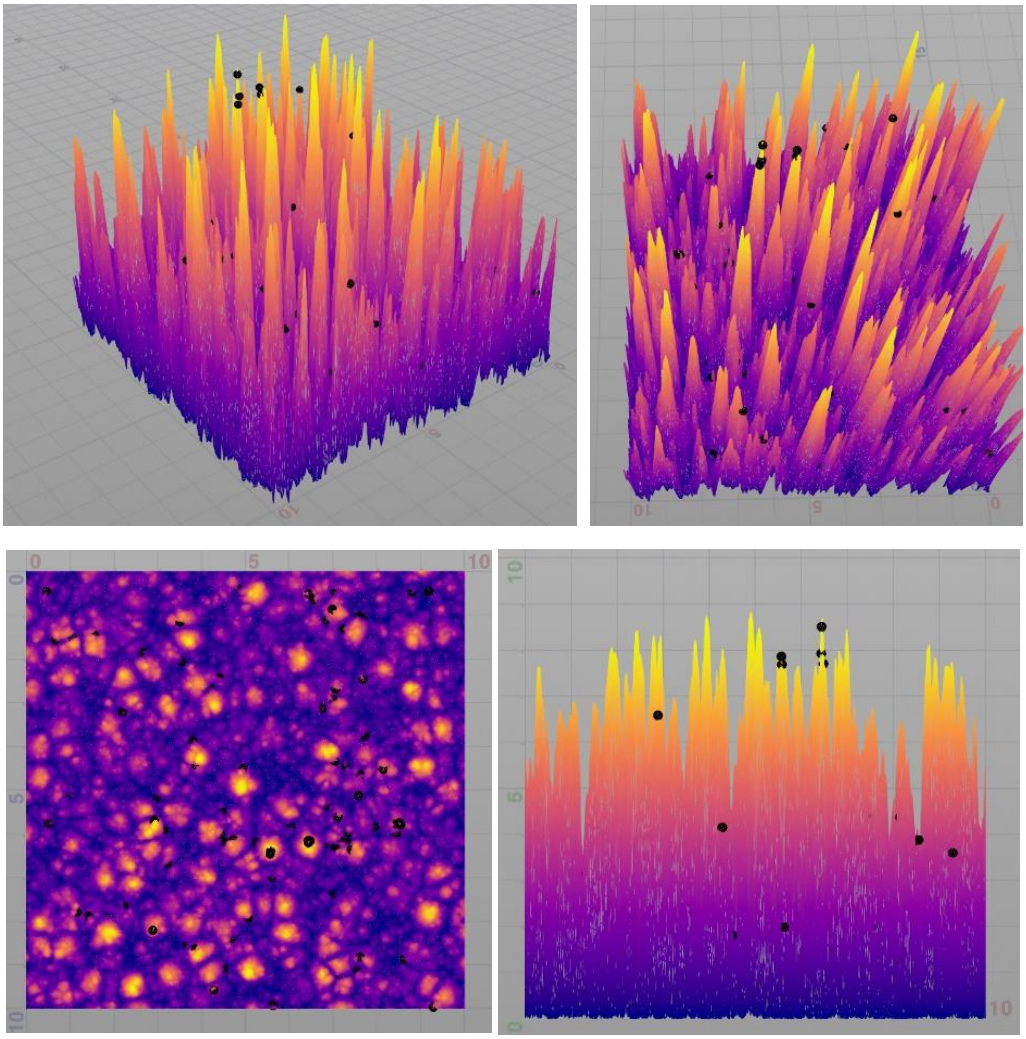

Image 5.17: Objective function, example 3

We start by testing pc=0.65, pm=0.15, sigma=0.25 (Image 5.18). We observe that the algorithm quickly converges to one of the local maxima, and due to the small σ, it cannot escape from it. For larger σ (e.g., σ=1), as seen in Image 5.19, the algorithm again converges toward the same local maximum, but the mutated members of the population are spread across a larger area.

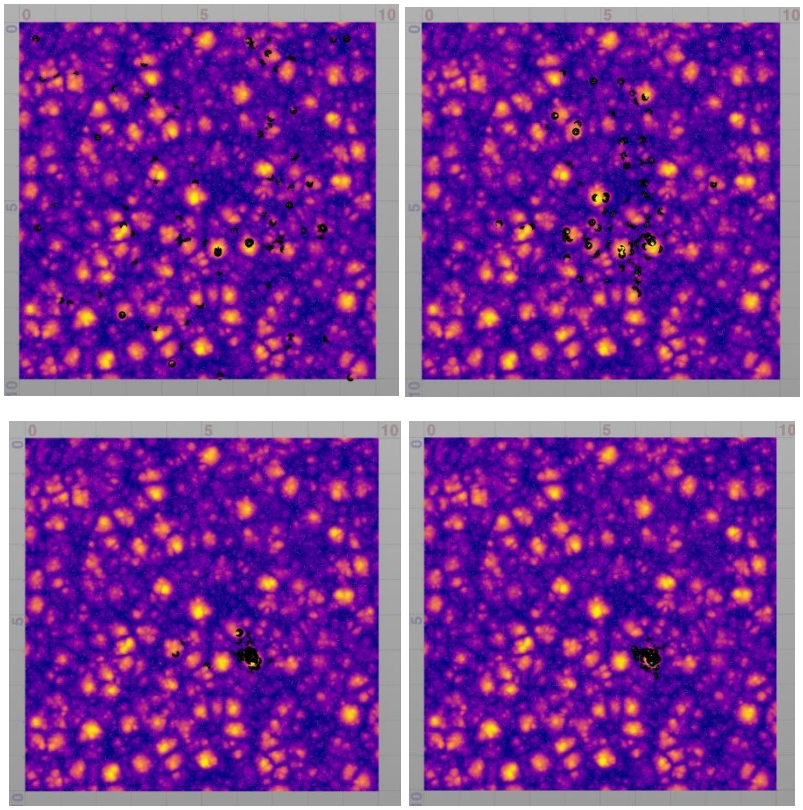

Image 5.18: Generations 1, 7, 14, 30 are shown in the top left, top right, bottom left, and bottom right, respectively. Parameters: pop=100, seed=1, pc=0.65, pm=0.15, sigma=0.25, elitism=0

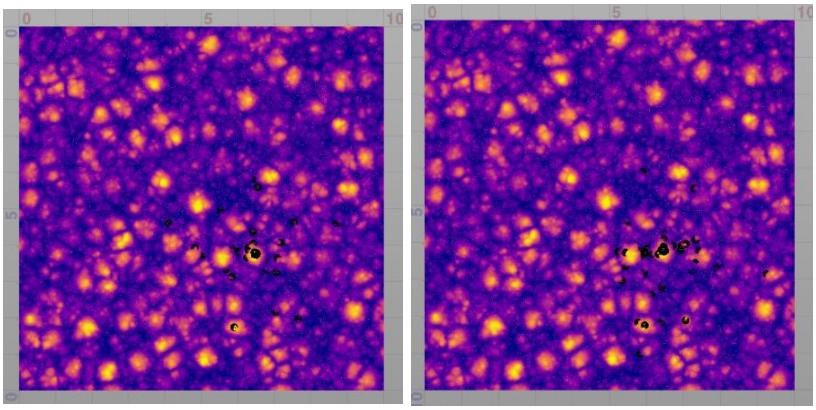

Image 5.19: Generations 25, 55 left and right respectively. Parameters: pop=100, seed=1, pc=0.65, pm=0.15, sigma=1, elitism=0

Let’s now try the inverse parameter combination, high pm and low pc, specifically pc=0.15, pm=0.75 with sigma=0.25 (Image 5.20). We observe that, at least for the first 25 generations, the algorithm's population is split among several local maxima. After examining the different local maxima, the algorithm settled on the best of them, and after 50 generations, the population is almost entirely at the global maximum of the objective function. Generally, we conclude that high values of pc increase the algorithm’s convergence speed but also increase the likelihood of prematurely converging to a local maximum. On the other hand, high values of pm enhance the

algorithm's ability to search for extrema in exchange for more computational time. It should be noted that achieving this performance also requires an appropriate σ. For example, if $\sigma = 1$ (Image 5.21), the algorithm fails to find any satisfying solutions, and the noise from the mutation dominates the population.

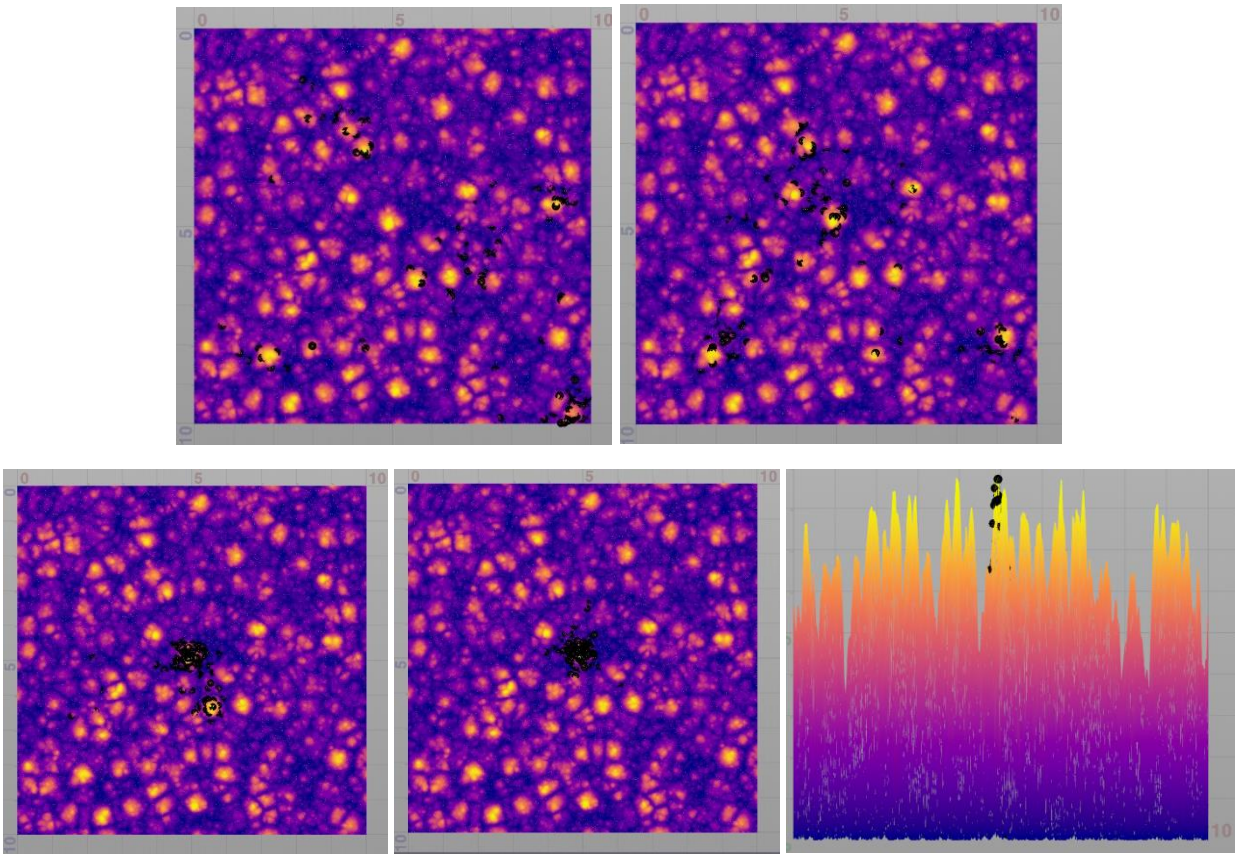

Image 5.20: Generations 10, 25, 50, 65 are shown in the top left, top right, bottom left bottom middle and bottom right, respectively. Parameters: pop=100, seed=1, pc=0.15, pm=0.75, sigma=0.25, elitism=0

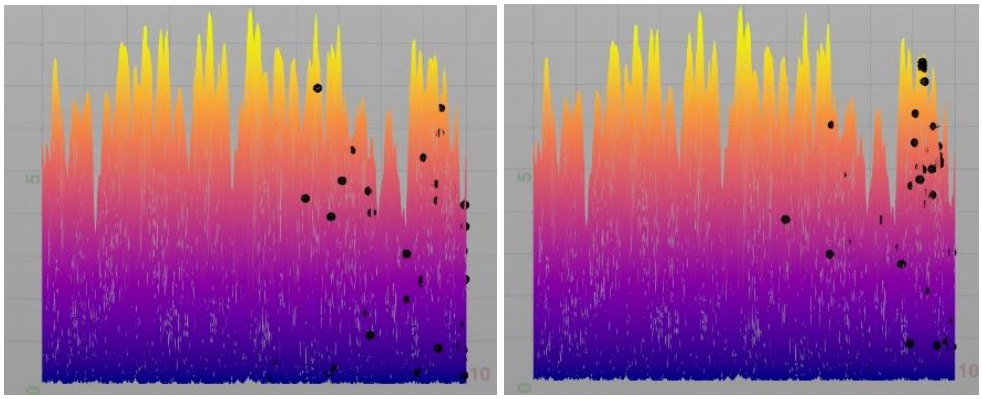

Image 5.21: Generations 60, 100 left and right respectively: pop=100, seed=1, pc=0.15, pm=0.75, sigma=1, elitism=0

As a final example of an objective function, we choose a function derived from Perlin noise, which, however, changes over time. Image 5.22 shows the evolution of the function over the course of 15 generations. This problem falls under the category of dynamic optimization problems. Without delving too deeply into the details, we will demonstrate that a GA with appropriate parameters can find the optima of the function as it changes. Based on the conclusions above, we will select a population of 100 members, pc=0.75 so that the algorithm converges quickly and keeps up with the changes in the function, pm=0.4 so the algorithm can find new extrema without the noise becoming dominant, sigma=1.6 so the population can detect new extrema even at a significant distance, and finally, active elitism so that the best solution up to that point is always preserved. In Images 5.23 and 5.24, we can see that the algorithm indeed continuously achieves a high maximum fitness by identifying local maxima.

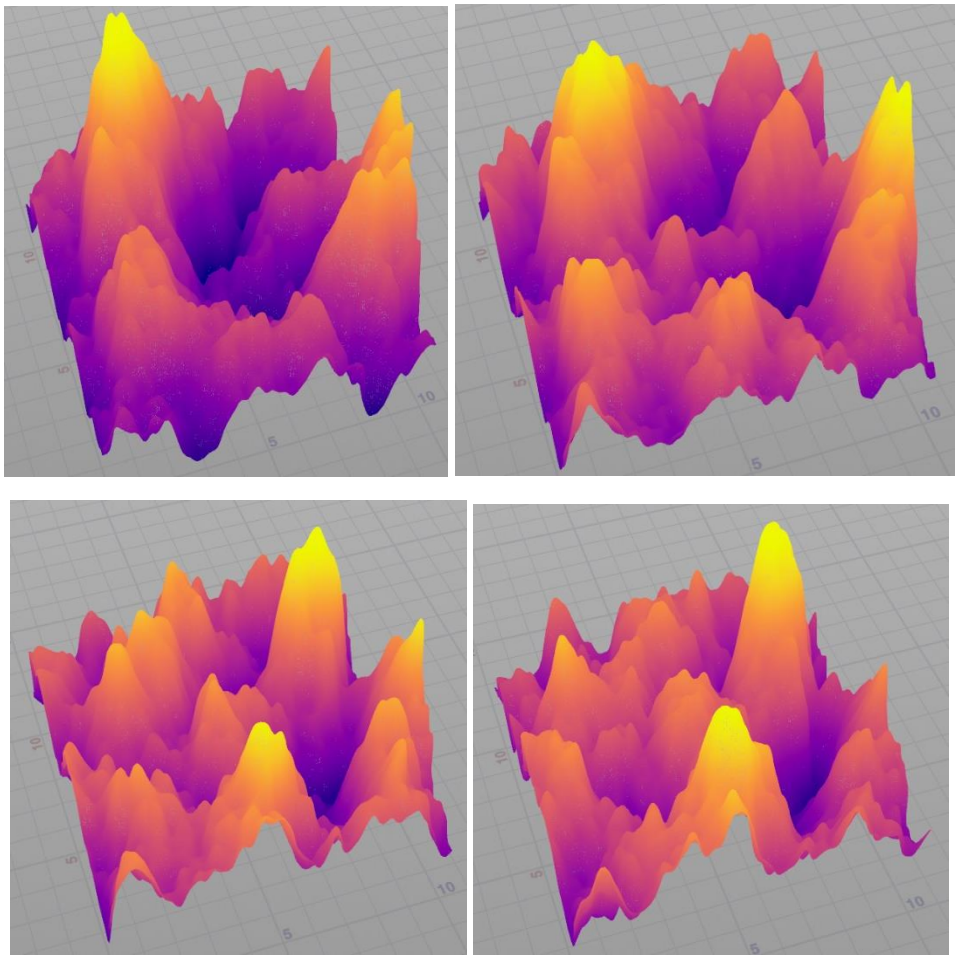

Image 5.22: Objective function, example 4. Generations 120, 125, 130, 135 are shown in the top left, top right, bottom left, and bottom right, respectively

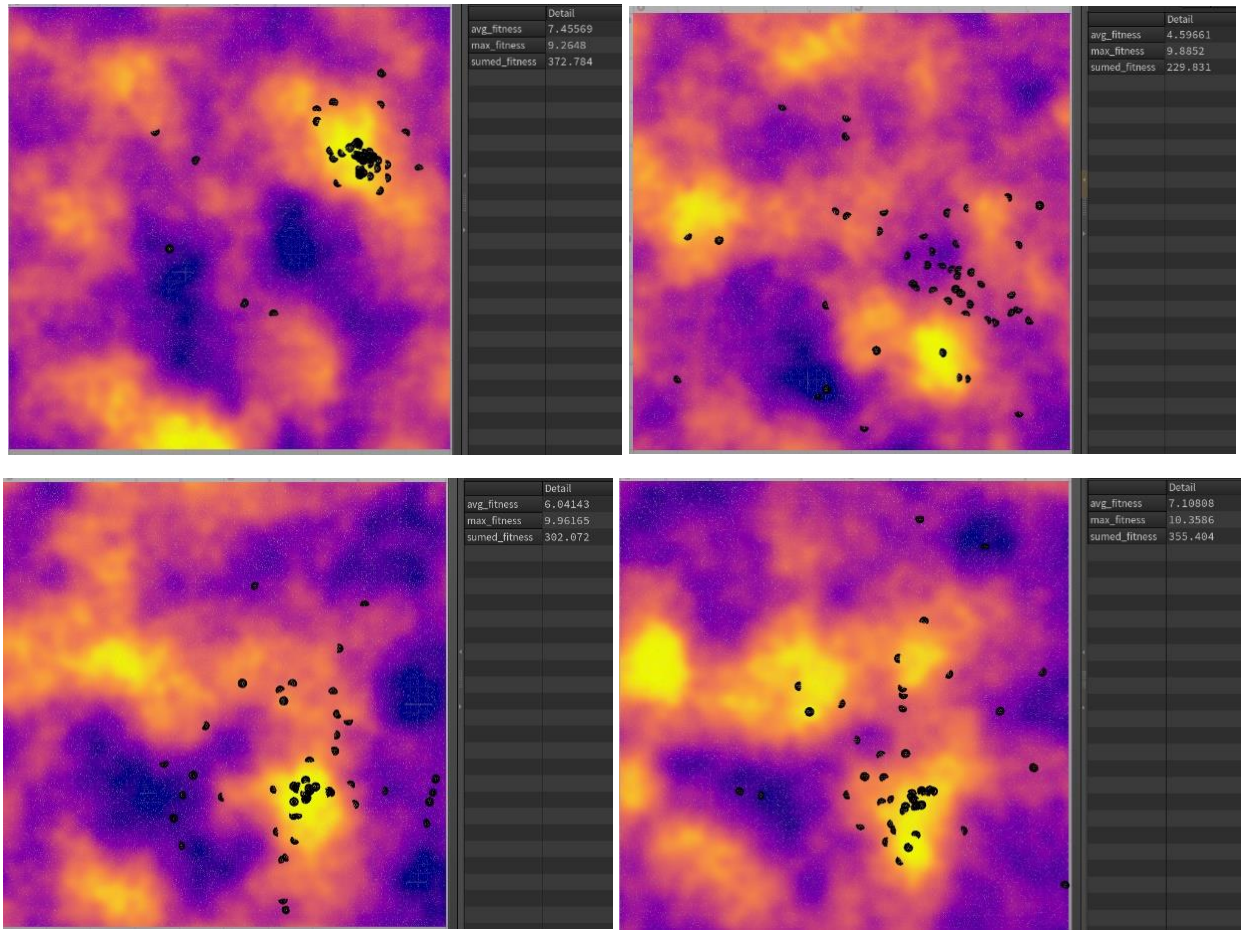

Image 5.23: Generations 125, 140, 155, 170 are shown in the top left, top right, bottom left, and bottom right, respectively. Parameters: pop=100, seed=1, pc=0.75, pm=0.4, sigma=1.6, elitism=1

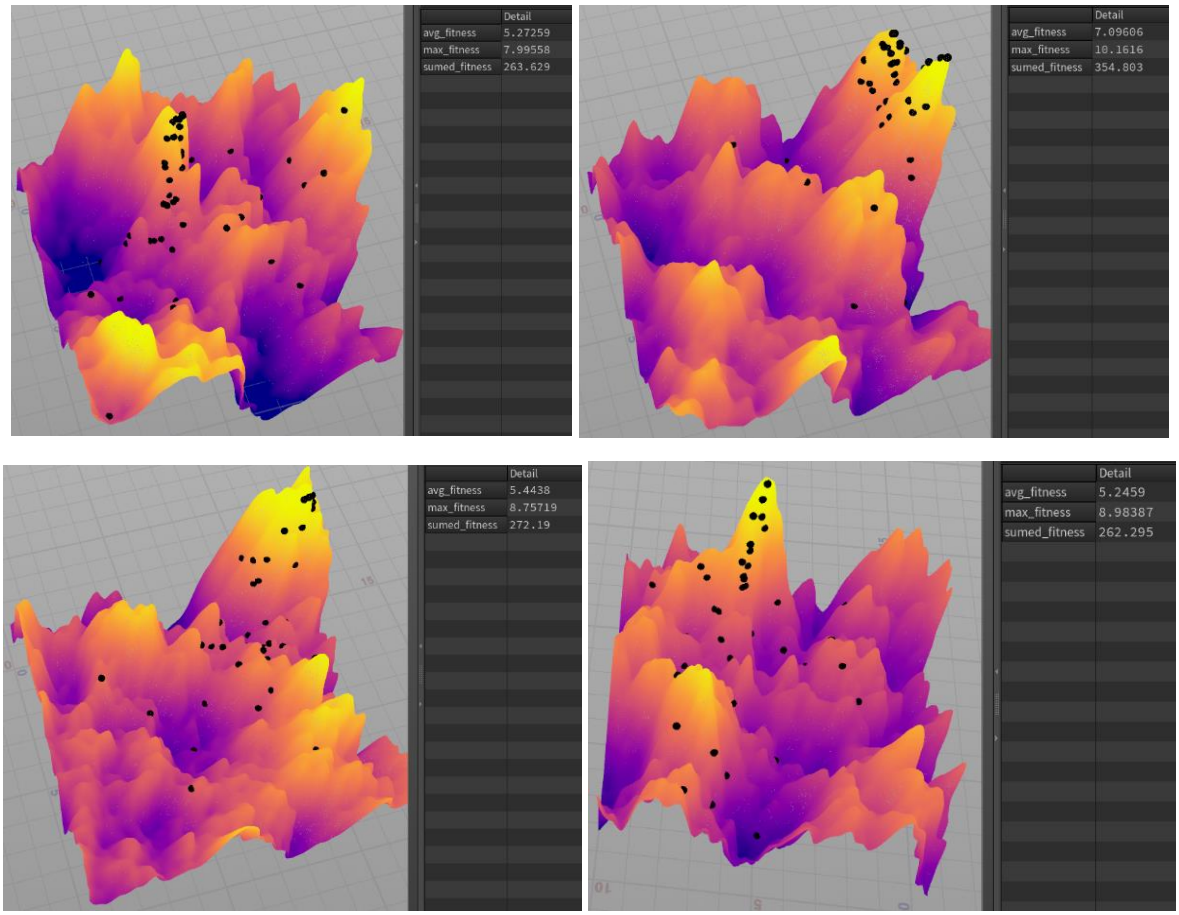

Image 5.24: Generations 210, 225, 240, 255 are shown in the top left, top right, bottom left, and bottom right, respectively. Parameters: pop=100, seed=1, pc=0.75, pm=0.4, sigma=1.6, elitism=1

Chapter 6 Conclusions – Extensions

In this work, we introduced genetic algorithms, examined some of their variations, and analyzed their operation and performance on specific examples. From these examples, as mentioned below, we can draw some conclusions. Finally, we briefly mention some additional extensions and variations for which more information can be found in the literature ([3], [7], [8]). These extensions can be incorporated into both the code provided in the appendix and the application in Chapter 5 aiming for their visualization.

In Chapter 2, we explored the parameters of the algorithm from Chapter 1 for Example 1. Through analysis, we identified some optimal parameters for this specific problem. In the following two chapters, we studied the variations of Gray encoding and the Real-valued Genetic Algorithm, performing similar parameter analysis as in Chapter 2, and determining some optimal parameters. We found that, despite the theoretical advantage of Gray encoding, it does not offer any practical benefit in this particular example. Moreover, the real-valued genetic algorithm did not seem to improve performance either. However, the problem in Example 1 is a very simple optimization problem. In problems with more parameters, constraints, and a more "exotic" landscape for the objective function, the three variations might exhibit significant differences in performance. We also reached this conclusion in Chapter 5, where we observed that different objective functions benefit from different parameters and variations of the algorithm. We found that a larger population size provides better results, albeit at the cost of computational time, with greater consistency in search spaces with many extrema. We also explored the effect of crossover probability on convergence speed and the mutation probability on the algorithm's ability to scan and explore the search space. Additionally, we examined the impact of certain parameters on the algorithm, such as binary string length in the basic algorithm and mutation standard deviation in RGAs. Finally, we saw how a simple GA variation, elitism, functions as a parameter within the algorithm and its effect on the best solutions found up to that point.

Optimizing the parameters of a GA is generally a process specific to a particular problem or a category of related problems. Determining which functions are "related" and which properties define this relationship is a question whose answer would enable the use of algorithms tailored to one problem for a broader category of problems. The pseudo-optimization process we followed for algorithm parameters is known as Grid Search, which involves scanning specific values and comparing the results they produce. We refer to it as pseudo-optimization because there was no precise objective function, but rather we analyzed performance metrics intuitively. Optimizing hyperparameters such as $p_\delta, p_\mu,$ is a field in its own right. It is worth noting that genetic algorithms have a place in this field as well, where they are known as meta-GAs. This refers to optimizing the parameters of a GA, another algorithm in general, a statistical model, etc., using a GA ([13], [14]).

Lastly, Adaptive Genetic Algorithms (AGAs) introduce even more complexity to the issue of selecting hyperparameters, as in these algorithms, the parameters change over time. For example, the algorithm might start with a high value of p_μ, which decreases as generations progress. Based on the conclusions we have drawn, the algorithm would initially explore various solutions, but

after a few generations, its ability to search for extrema would diminish, settling on one. Such approaches obviously introduce many new capabilities to a GA but also new hyperparameters related to the rate of change and the range of values of a variable hyperparameter ([6]).

Of course, there are simpler variations of genetic algorithms for one to try. Most of these begin by questioning and then modifying a part of the basic algorithm. For instance, during the crossover stage, instead of selecting one crossover point between chromosomes, one could opt for two or more points. For two points, this method is called 2-point Crossover. Generally, the crossover, as we have seen it, is a convenient special case of uniform crossover. In this method, we use a binary string of the same length as the chromosomes, called a mask. This mask determines which bits will be exchanged between the two chromosomes undergoing crossover: for any bit in the mask that is 1, the corresponding bits of the selected chromosomes are exchanged, while if it is 0, they remain unchanged. Table 6.1 schematically shows both methods. Essentially, uniform crossover generalizes both other methods: single-point crossover is uniform crossover with a mask of the form 0000011111, while 2-point crossover, for example, can be represented by a mask of the form 0001111000 for chromosomes of length L=10. Syswerda (1989) [7] argued that 2-point crossover consistently outperforms single-point crossover, while uniform crossover is better than both in combining the information carried by the chromosomes.

Selected Chromosomes	New Chromosomes after Crossover
2-point Crossover	
101001|**1010**|110	101001|**1101**|110
011110|**1101**|010	011110|**1010**|010
Uniform Crossover	
Mask: 1110111000101	
1010**011**010**11**0	0110101010010
0111**101**101**0**1**0**	1011011101110

Table 6.1: Examples of other crossover methods

For the chromosome selection stage, many alternatives have been suggested to roulette selection. The selection stage is of great importance, as it is the step in which it is determined which chromosomes are more dominant, and their superiority over others is quantified with the goal of selecting them. A selection method can give emphasis, for example, to the individual with the highest fitness. The variation of elitism, which we discussed earlier, quantitatively gives "infinite" superiority to the individual with the highest fitness over the others. In Chapter 1, the choice of c in Figure 1.5 assigns different selection probabilities to the members of the population, increasing or decreasing their dominance. Briefly, we mention two other popular methods: tournament selection and ranking selection. In tournament methods, random pairs of population members are selected. From these pairs, the member with the higher fitness is chosen for crossover with a probability $p > 0.5$. In ranking methods, the chromosomes are ranked in descending order based on fitness. The member with the highest fitness is assigned rank 1, the second-highest fitness member is assigned rank 2, and so on. This ranking can be used in various ways, but the general category is referred to as “Ranking methods”. [3]

Finally, since we have extensively discussed GA efficiency, it is worth mentioning Parallel Genetic Algorithms (PGAs). As noted in the advantages of GAs in Chapter 1, the calculations for the objective function are independent of each other, leading to the straightforward assumption that they can be parallelized. Based on this observation, the first of three categories of PGAs is constructed, known as the Master-slave type. We will briefly cover all three categories. In the Master-slave algorithm, one processing unit manages the overall algorithm, handling crossover, mutation, and the execution of the main program, while the remaining processing units/nodes are responsible for computing the objective function for each population member. This approach is the simplest way to parallelize a GA and does not change the algorithm's functionality. It also addresses the issue of high computational cost that an objective function might incur. The second category of PGAs (Coarse-Grained GAs) modifies the algorithm's functionality to take advantage of a parallel computing system. It divides the population into subpopulations and runs the algorithm on these subgroups. Periodically, the algorithm allows information exchange between the subpopulations, for instance, by exchanging a random or dominant individual. Without this step, it would be as if several independent GAs were running on the same problem simultaneously. According to the literature [7], they are faster than the previous category and GAs with a single population. The third category of PGAs (Fine-Grained GAs) divides the population into very small subpopulations, often with only one member each. Each node contains a subpopulation that can interact with "neighboring" nodes. The "neighborhood" can be defined as a simple Cartesian grid or based on the topology most suitable for the architecture of the parallel system. [1], [7]

Presenting just a few more variations and options for the operation of genetic algorithms at a surface level reveals the flexibility they offer as tools for problem-solving. Depending on the difficulty, the nature of the problem, our requirements, and the computational systems available, we can choose the GA that best suits our needs. As we saw in the examples, a GA variation with certain hyperparameters is not necessarily efficient for solving a particular optimization problem. This result is also confirmed by the No Free Lunch (NFL) theorem [15], which states that satisfactory performance of an optimization algorithm on one family of problems does not guarantee its performance on another group of problems. However, the existence of so many variations and the ability to fine-tune algorithms through hyperparameters allows the broader family of genetic algorithms to successfully tackle many different problems in various scientific fields.

References

[1] Cantú-Paz, E. (1998). A Survey of Parallel Genetic Algorithms, Department of Computer Science University of Illinois

[2] Chakraborty, U.K. & Janikow, C.Z. (2003). An Analysis of Gray versus Binary Encoding in Genetic Search, Department of Mathematics and Computer Science University of Missouri

[3] Coley, D.A. (1999), An Introduction to Genetic Algorithms for Scientists and Engineers, World Scientific Publishing

[4] Frederick, W.G., Sedlmeyer, R.L. & White, C.M. (1993). The Hamming Metric in Genetic Algorithms and Its Application to Two Network Problems, Computer Science Department Indiana University

[5] Goldberg, D.E. (1989). Genetic Algorithms in Search, Optimization & Machine Learning, Addison-Wesley Publishing Company

[6] Han, S. & Xiao, L. (2022). An improved adaptive genetic algorithm, Department of Computer Sciencet, Ezhou University

[7] Haupt, R.L., Haupt, S.E. (2004). PRACTICAL GENETIC ALGORITHMS, A Wiley-Interscience publication

[8] Mitchell, M. (1999). An Introduction to Genetic Algorithms, A Bradford Book The MIT Press

[9] Βασιλείου, Π.-Χ.Γ. & Γεωργίου, Α.Κ. (1993). Μη γραμμικές μέθοδοι βελτιστοποίησης, Εκδόσεις Ζήτη

[10] Cellular Noise, https://thebookofshaders.com/12/

[11] Genetic Algorithm, https://en.wikipedia.org/wiki/Genetic_algorithm

[12] Gray to Binary and Binary to Gray conversion, https://www.geeksforgeeks.org/gray-to-binary-and-binary-to-gray-conversion/

[13] Hyperparameter Optimization, https://en.wikipedia.org/wiki/Hyperparameter_optimization

[14] Meta Optimization, https://en.wikipedia.org/wiki/Meta-optimization

[15] No Free Lunch in Search and Optimization, https://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization

[16] Perlin Noise: A Procedural Generation Algorithm, https://rtouti.github.io/graphics/perlin-noise-algorithm

Software:

- Python 3.12, https://www.python.org/
- HoudiniFX 20.0 https://www.sidefx.com/products/houdini/
- Desmos Graphing Calculator https://www.desmos.com/

Appendix: Code

Program 1

```
import numpy as np
import random as rd
import copy as cp
import time
#program 2 is gray.py
#import from gray.py binary2gray, gray2binPopulation
from gray import binary2gray, gray2binPopulation

G = 10  #total generations
N = 8  #number of chromosomes per generation. Even number
L = 9  #binary length
C_MINMAX_FIT_PERCENTAGE = 0.01
Cross_prob = 0.8  #probability of crossover
Mutation_prob = 0.01 #probability of mutation
a = -15  #domain [a,b]
b = 15
GrayEncoding = 0  #0 for binary, 1 for gray
rd.seed(1)


#calculation of a function f(x)
def f(x):
  f_at_x = 2 * np.sin(x) + x + 0.1 * pow(x - 5.5, 2)  # define function
here
  return f_at_x


#Population initialization
def init_population():
  Population = []
  for c in range(N):
    chromosome = ''
    for j in range(L):
      bit = str(rd.randint(0, 1))
      chromosome = chromosome + bit
    Population.append(chromosome)
  return Population


#converts binary string list to integer list
def binStr2int_List(pop):
  inpt = cp.deepcopy(pop)
  for i in range(N):
```

```
    inpt[i] = int(inpt[i], 2)
  return inpt


#converts binary string list to float list in [a, b]
def intRemap_List(pop):
  inpt = cp.deepcopy(pop)
  inpt = binStr2int_List(inpt)
  nom = b - a
  den = pow(2, L) - 1
  frac = nom / den
  for i in range(N):
    inpt[i] = a + frac * inpt[i]
  return inpt


#calculates fitness of each chromosome, input float list
def fitness(pop):
  inpt = cp.deepcopy(pop)
  inpt = intRemap_List(inpt)
  for i in range(N):
    inpt[i] = f(inpt[i])
  return inpt


#compute average fitness of population
def avg_fitness(pop):
  inpt = cp.deepcopy(pop)
  inpt = fitness(inpt)
  avg = np.mean(inpt)
  return avg


#compute diversity of the population by summing the hamming distance of
each chromosome
def normalized_ham_diversity(pop):
  inpt = cp.deepcopy(pop)
  sum_distances = 0
  for i in range(N):
    for j in range(i + 1, N):
      distance = 0
      for k in range(L):
        if inpt[i][k] != inpt[j][k]:
          distance += 1
      sum_distances += distance / L
```

```
  norm = 0.5 * N * (N - 1)
  norm_sum_distances = sum_distances / norm
  return norm_sum_distances


#selects a chromosome for reproduction. Returns index of chromosome
def selection(pop):
  inpt = cp.deepcopy(pop)
  fit = fitness(inpt)
  c = C_MINMAX_FIT_PERCENTAGE * (max(fit) - min(fit))
  inv_fit = max(fit) + c - fit
  inv_fit_sum = sum(inv_fit)
  prob = []
  for i in range(N):
    if inv_fit_sum != 0:
      prob.append(inv_fit[i] / inv_fit_sum)
    else:
      #if i == 0:
      #print("total population convergence")
      prob.append(1 / N)

  i = N
  while (i >= N):  #in case of accumulation of float errors: repeat
selection
    selector = rd.random()
    prob_sum = prob[0]
    i = 0
    #print(selector)
    #print(fit)
    #print(prob)
    #print(sum(prob))
    while (selector > prob_sum):  #selection loop
      prob_sum = prob_sum + prob[i]
      #print(prob_sum)
      i = i + 1
      #print(i)
  return i


#crossover over 2 binary strings
def crossover(bin1, bin2):
  foo = rd.random()
  if (foo < Cross_prob):
    cutpoint = rd.randint(1, L - 1)
    #print(cutpoint)
```

```
    child1 = bin1[0:cutpoint] + bin2[cutpoint:L]
    child2 = bin2[0:cutpoint] + bin1[cutpoint:L]
    bin1 = child1
    bin2 = child2
  return [bin1, bin2]


#mutation of a binary string
def mutation(bin):
  bar = list(bin)
  for i in range(L):
    foo = rd.random()
    if (foo < Mutation_prob):
      bar[i] = str(1 - int(bar[i]))
  bin = "".join(bar)

  return bin


#combination of selection, crsossover and mutation over a bin string
population
def evolve1generation(pop):
  inpt = cp.deepcopy(pop)
  sel_chroms = []
  nextgen = []
  for i in range(N):
    j = selection(inpt)  #select individuals
    if (GrayEncoding == 1):
      sel_chroms.append(binary2gray(inpt[j]))
    else:
      sel_chroms.append(inpt[j])
    if (i % 2 == 1):
      next2 = crossover(sel_chroms[i - 1], sel_chroms[i])  #crossover
      for j in range(2):
        mutated = mutation(next2[j])  #mutation
        nextgen.append(mutated)
  if (GrayEncoding == 1):
    nextgen = gray2binPopulation(nextgen)
  return (nextgen)


def example1():
  print('change algorithm parameters in line 5')
  print('function can be changed in line 18')
```

```
  Population = init_population()
  for i in range(G - 1):
    print('--Generation: ', i, '--', sep='')
    x_points = intRemap_List(Population)
    #print([ '%.2f' % elem for elem in x_points ])
    print('Average Fitness: ', avg_fitness(Population))
    print('Diversity: ', normalized_ham_diversity(Population))
    print('Best Solution: ', min(fitness(Population)))
    Population = evolve1generation(Population)
  print('--Last Generation: ', G - 1, '--', sep='')
  print(Population)
  print('Average Fitness: ', avg_fitness(Population))
  print('Diversity: ', normalized_ham_diversity(Population))
  print("Best Solution: ", min(fitness(Population)))


def example2():
  G_list = [5, 10, 15, 25, 50]  #total generations
  N_list = [4, 8, 12, 16,
            20]  #number of chromosomes per generation. Even number
  L_list = [5, 6, 9, 12, 15, 20]  #binary length
  C_prob_list = [1, 0.8, 0.7, 0.5, 0.3]  #probability of crossover
  M_prob_list = [0, 0.01, 0.05, 0.1, 0.3]  #probability of mutation
  #average results for number of reps
  reps = 1000
  for x in L_list:
    global L
    L = x
    avg = gen = div = min_fit = 0
    start_time = time.time()
    for j in range(reps):
      rd.seed(88 * j + 2024)
      Population = init_population()
      a = avg_fitness(Population)
      d = normalized_ham_diversity(Population)
      m = min(fitness(Population))
      g = 0
      #find best fitness and avg fitness, diversity, generation when
achieved
      for i in range(G - 1):
        Population = evolve1generation(Population)
        if (min(fitness(Population)) < m):
          g = i
          #print(i)
          a = avg_fitness(Population)
```

```
            d = normalized_ham_diversity(Population)
            m = min(fitness(Population))
        #averaging results
        avg = avg + a
        div = div + d
        min_fit = min_fit + m
        gen = gen + g
        res = [
            round(min_fit / reps, 2),
            round(avg / reps, 2),
            round(div / reps, 2),
            round(gen / reps, 2)
        ]
        end_time = time.time()
        elapsed_time = round(end_time - start_time, 2)
      print(G, N, L, Cross_prob, Mutation_prob, ":", res, "in",
elapsed_time,
            "sec")


def example3():
  G_list = [5, 10, 15, 25, 50]  #total generations
  N_list = [4, 8, 12, 16, 20]  #number of chromosomes per generation.
Even number
  L_list = [6, 9, 12, 15, 20]  #binary length
  C_prob_list = [1, 0.8, 0.7, 0.5, 0.3]  #probability of crossover
  M_prob_list = [0, 0.01, 0.05, 0.1, 0.3]  #probability of mutation
  #average results for number of reps
  reps = 10
  Results = []
  Parameters = []
  for G in G_list:
    print("G=", G, "***")
    for n in N_list:
      global N
      N = n
      print("N=", N, "**")
      for l in L_list:
        global L
        L = l
        print("L=", L)
        for c in C_prob_list:
          global Cross_prob
          Cross_prob = c
          for m in M_prob_list:
```

```
            global Mutation_prob
            Mutation_prob = m
            avg = gen = div = min_fit = 0
            for j in range(reps):
              rd.seed(520 * j + 68)
              Population = init_population()
              a = avg_fitness(Population)
              d = normalized_ham_diversity(Population)
              m = min(fitness(Population))
              g = 0
              #find best fitness and avg fitness, diversity, generation
when achieved
              for i in range(G - 1):
                Population = evolve1generation(Population)
                if (min(fitness(Population)) < m):
                  g = i
                  #print(i)
                  a = avg_fitness(Population)
                  d = normalized_ham_diversity(Population)
                  m = min(fitness(Population))
              #averaging results
              avg = avg + a
              div = div + d
              min_fit = min_fit + m
              gen = gen + g
              res = [
                  round(min_fit / reps, 2),
                  round(avg / reps, 2),
                  round(div / reps, 2),
                  round(gen / reps, 2)
              ]
            Results.append(res)
            foo = [G, N, L, Cross_prob, Mutation_prob]
            Parameters.append(foo)
  #List of lists of GA parameters
  print(Parameters)
  #List of lists of GA metrics. Parameters[i] corresponds to Results[i]
  print(Results)
  print(len(Parameters), "elements")


def main():
  example2()
```

Program 2

```
# Based on the code from user mits. Source:
# https://www.geeksforgeeks.org/gray-to-binary-and-binary-to-gray-
conversion/
def xor_c(a, b):
  return '0' if (a == b) else '1'


def flip(c):
  return '1' if (c == '0') else '0'


def binary2gray(binary):
  gray = ""
  gray += binary[0]
  for i in range(1, len(binary)):
    gray += xor_c(binary[i - 1], binary[i])
  return gray


def gray2binary(gray):
  binary = ""
  binary += gray[0]
  for i in range(1, len(gray)):
    if (gray[i] == '0'):
      binary += binary[i - 1]
    else:
      binary += flip(binary[i - 1])
  return binary

def bin2grayPopulation(pop):
  grayPop = []
  for i in range(len(pop)):
    grayPop.append(binary2gray(pop[i]))
  return grayPop

def gray2binPopulation(grayPop):
  binPop = []
  for i in range(len(grayPop)):
    binPop.append(gray2binary(grayPop[i]))
  return binPop
```

Program 3

```
import numpy as np
import random as rd
import copy as cp
import time


G = 10  #total generations
N = 8  #number of chromosomes per generation. Even number
sigma = 1
C_MINMAX_FIT_PERCENTAGE = 0.01
Cross_prob = 0.8  #probability of crossover
Mutation_prob = 0.2 #probability of mutation
a = -15  #domain [a,b]
b = 15
rd.seed(3)

#calculation of a function f(x)
def f(x):
f_at_x = 2 * np.sin(x) + x + 0.1 * pow(x - 5.5, 2)  # define function
here
return f_at_x

#Population initialization
def init_population():
pop = []
for i in range(N):
      pop.append(rd.uniform(a, b))
return pop

#calculates fitness of each chromosome, input float list
def fitness(pop):
inpt = cp.deepcopy(pop)
for i in range(N):
      inpt[i] = f(inpt[i])
return inpt

#compute average fitness of population
def avg_fitness(pop):
inpt = cp.deepcopy(pop)
inpt = fitness(inpt)
avg = np.mean(inpt)
return avg

#selects a chromosome for reproduction. Returns index of chromosome
def selection(pop):
inpt = cp.deepcopy(pop)
fit = fitness(inpt)
c = C_MINMAX_FIT_PERCENTAGE * (max(fit) - min(fit))
```

```
inv_fit = max(fit) + c - fit
inv_fit_sum = sum(inv_fit)
prob = []
for i in range(N):
      if inv_fit_sum != 0:
            prob.append(inv_fit[i] / inv_fit_sum)
      else:
            #if i == 0:
            #print("total population convergence")
            prob.append(1 / N)

i = N
while (i >= N):  #in case of accumulation of float errors: repeat
selection
      selector = rd.random()
      prob_sum = prob[0]
      i = 0
      #print(selector)
      #print(fit)
      #print(prob)
      #print(sum(prob))
      while (selector > prob_sum):  #selection loop
            prob_sum = prob_sum + prob[i]
            #print(prob_sum)
            i = i + 1
            #print(i)
return i

#crossover via blending over 2
def blend(x1, x2):
if (rd.random() < Cross_prob):
      b1 = rd.random()
      b2 = rd.random()
      y1 = b1 * x1 + (1 - b1) * x2
      y2 = b2 * x1 + (1 - b2) * x2
else:
      y1 = x1
      y2 = x2
return [y1, y2]

#mutation of a chromosome
def mutation(x):
if (rd.random() < Mutation_prob):
      x = x + sigma*rd.uniform(-1, 1)
      if x > b or x < a:
            x = rd.uniform(a, b)
return(x)
```

```
#combination of selection, crsossover and mutation over a bin string
population
def evolve1generation(pop):
inpt = cp.deepcopy(pop)
sel_chroms = []
nextgen = []
for i in range(N):
      j = selection(inpt)  #select individuals
      sel_chroms.append(inpt[j])
      if (i % 2 == 1):
            next2 = blend(sel_chroms[i - 1], sel_chroms[i])  #crossover
            for j in range(2):
                  mutated = mutation(next2[j])  #mutation
                  nextgen.append(mutated)
return (nextgen)

def example1():
print('change algorithm parameters in line 5')
print('function can be changed in line 18')

Population = init_population()
for i in range(G - 1):
      print('--Generation: ', i, '--', sep='')
      #print([ '%.2f' % elem for elem in x_points ])
      print('Average Fitness: ', avg_fitness(Population))
      print('Best Solution: ', min(fitness(Population)))
      Population = evolve1generation(Population)
print('--Last Generation: ', G - 1, '--', sep='')
print(Population)
print('Average Fitness: ', avg_fitness(Population))
print("Best Solution: ", min(fitness(Population)))

def example2():
G_list = [5, 10, 15, 25, 50]  #total generations
N_list = [4, 8, 12, 16,
                              20]  #number of chromosomes per
generation. Even number
sigma_list = [0.1, 0.5, 1, 2, 3]  #mutation standard deviation
C_prob_list = [1, 0.8, 0.7, 0.5, 0.3]  #probability of crossover
M_prob_list = [0, 0.1, 0.2, 0.3, 0.5]  #probability of mutation
#average results for number of reps
reps = 100
for x in M_prob_list:
      global Mutation_prob
      Mutation_prob = x
      avg = gen = div = min_fit = 0
      start_time = time.time()
      for j in range(reps):
            rd.seed(77 * j + 2024)
```

```
            Population = init_population()
            a = avg_fitness(Population)
            m = min(fitness(Population))
            g = 0
            #find best fitness and avg fitness, diversity, generation
when achieved
            for i in range(G - 1):
                  Population = evolve1generation(Population)
                  if (min(fitness(Population)) < m):
                        g = i
                        #print(i)
                        a = avg_fitness(Population)
                        m = min(fitness(Population))
            #averaging results
            avg = avg + a
            min_fit = min_fit + m
            gen = gen + g
            res = [
                        round(min_fit / reps, 2),
                        round(avg / reps, 2),
                        round(gen / reps, 2)
            ]
            end_time = time.time()
            elapsed_time = round(end_time - start_time, 2)
      print(G, N, sigma, Cross_prob, Mutation_prob, ":", res, "in",
elapsed_time,
                        "sec")

def main():
example2()

if __name__ == "__main__":
main()
```