# High probability mutation in Genetic Algorithms

*Nicolae-Eugen Croitoru*

"Alexandru Ioan Cuza" University of Iași

September 2016

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I would like to thank my supervisor, Prof. Henri Luchian, PhD., and everyone who participated in the review of my work.

I owe special thanks to an anonymous reviewer who, during the SYNASC 2014 conference review process, suggested investigating Error Thresholds in the context of High-Probability Mutation.

# Summary

This work contains an investigation of high mutation probabilities ($\approx 0.95$) used in Genetic Algorithms. The comparison to low probability mutation constitutes the basic general structure of the experiments performed and arguments presented. Benchmark numerical optimisation functions, as well as the Royal Road Function are used to compare the two mutation types, and deduce the properties of high-probability mutation from the differences.

Determining Error Thresholds helps show that high-probability mutation in Genetic Algorithms is not a Random Search-type Algorithm, but an evolutionary one. Using Consensus Sequence Plots as a graphical tool, Error Thresholds are visually presented, in the context of changing mutation probabilities.

A prediction problem using real-world data is approached by evolving Artificial Neural Networks. The addition of high-probability mutation is shown to lower the overfit and improve prediction quality.

Genetic Algorithm performance is shown to depend primarily - in this context - not on mutation probability, but on entropy induced by mutation into the population. High-probability mutation is found to be a low-entropy mutation which, by forming approximate dual representations every other generations, increases the exploratory tendencies of Genetic Algorithms. High-probability mutation performs better at escaping plateaus, and at preventing premature convergence and overfit; it is outperformed by low-probability mutation when those features are not needed.

# 1. Introduction

Genetic Algorithms[19] are a class of meta-heuristic search algorithms, inspired by the biological evolution by natural selection. They manage good search and optimisation performance on many types of problems, despite modelling the complex process of biological evolution in a relatively simple way.

Genetic Algorithms are a population-based method (as opposite to a trajectory-based one): they maintain a population of genomes (also named genotypes, individuals or, in haploid Genetic Algorithms, chromosomes - all these names are used interchangeably in this work). In the simplest case, all genomes have equal lengths, and their basic building blocks, nucleotide-analogues, are simple bits, 0 and 1.

A series of nature-analogous basic operators are used to simulate evolution on that population. Each genome expresses its genes, is evaluated according to that phenotype expression, and given a fitness. Individuals are treated as replicators, and are subjected to differential selection according to their relative fitness. The selection operator directs evolution, aiming it towards maximising the fitness. By choosing an appropriate phenotype expression function and an appropriate fitness measure, the Genetic Algorithm's search can be directed towards solving a problem.

Recombination (or cross-over) emulates simple sexual reproduction or bacterial gene transfer[23]; by allowing already-existing genes to permute from one genome configuration to another, favourable recombinations appear. Cross-over attempts to improve fitness by exploiting the information already available in the gene-pool.[1]

[1] In this work, cross-over is often shortened to *cx*.

Mutation, based on the biological phenomenon of replication error, induces random changes to genes; new genes, and thus, new traits can appear through this random process. The mutation operator, by randomly introducing new information into the population, explores new possible genome configurations. [2]

However, evolution takes time (emulated as the successive application of the different operators on the population). Simple attempts at satisfying the objective function - at obtaining a good fitness - are evolved, then replaced with better genomes - better potential solutions - over and over again. The combined information gained by evolution is stored within all the genomes in the population. New individuals and new generations rely on the old ones for their evolutionary improvement and optimisation.

A danger when using the mutation operator is setting the induced error rate too high; a high-enough mutation randomly mutates genes beyond the ability of the evolutionary process to correct or compensate for. Instead of evolving a population of one or several groups of mostly-similar individuals (or quasispecies[17]), the population degenerates in a group of divergent, dissimilar genomes. As genes are no longer kept mostly intact from parents to descendants, evolution cannot accumulate information, and the whole evolutionary process is disrupted. In biological evolution, this most often results in immediate cellular death; in Genetic Algorithms it results in Random Search[36].

While, in the context of the No Free Lunch Theorem[45], Random Search Algorithms have applications where they outperform Genetic Algorithms, hybridising the two by increasing the mutation rate is generally avoided.

However, in the context of Genetic Algorithms, a high mutation rate and evolution disruption are not synonymous. If the probability of mutating each gene in the population is very high (in the proximity of 1), due to the synchronised nature of mutation in a genetic algorithm (i.e. all operator actions happen at the same time: the mutations in a generation happen at once, not at random times) and the simple binary encoding of the

[2] Where appropriate, mutation is replaced with the letter $m$. For instance, mutation probability can appear as $p_m$.

representation, evolution will not be disrupted. Instead, if mutation bit-flips most genes in one generation, the vast majority of those genes will be bit-flipped back. By observing gene change every two generations, the effective changes brought by high-probability mutation can be shown to be relatively low.

This is the core argument of this dissertation: that high-probability mutation is a variant type of mutation in the context of Genetic Algorithms; it is not a Random Search Algorithm hybridisation.

Beyond theoretical arguments, low- and high-probability mutation are compared to each other, and contrasted with high-entropy, evolution-scrambling mutation. In addition to statistical measurements of Genetic Algorithm performance, population evolution is analysed in-depth: by using Consensus Sequence Plots[32], the thresholds between functioning and disrupted evolution are found. Such Error Thresholds under which evolution can happen undisrupted are found, even for high-probability mutation.

In addition to well-known optimisation benchmark problems, the characteristics of high-probability mutation are put to the test using real-world data. It is shown to lower overfit in a Neuroevolution algorithm attempting to predict the future state of an Internet-based social network.

## 1.1. Aim

The purpose of this work is to better understand high-probability mutation, and show it as worthwhile tool in the Genetic Algorithm toolbox. This understanding allows for stating a basic rule-of-thumb, to advise when high-probability mutation might be useful, and when it will likely be detrimental to a Genetic Algorithm.

Another objective is to show high-probability mutation in Genetic Algorithms is different from a Random Search Algorithm. This is accomplished in three ways:

- theoretically, by statistically computing event chains occurring during mutation.
- indirectly, by showing that low- and high-probability mutation have similar performances to each other, and different performance to high-entropy, random-search mutation.
- directly, by visually and statistically showing the formation of stable evolved structures within the population for both low- and high-probability mutation, where for high-entropy mutation they do not.

Additionally, it is desired for the current study to be as relevant as possible. Since the majority of Genetic Algorithms used at the time of writing are incremental or evolved derivatives of John Holland's initial method[19][3], the Genetic Algorithm used here is a close derivative.

It is hoped that results relevant to this algorithm will apply to many other GA variants and hybrids used or researched. Through a "phylogenetic metaphor", the root of the GA variant tree is expected to be most representative to the class, and, overall, the single variant most similar to all of them.

It is due to this desire for relevance that most tests are performed on well-known and widely-used benchmark function instances; at the same time, real-world data is used to test the method, to show its practical applicability.

[3] As shown by the comparatively high number of citations that dissertation received

## 1.2. Published Work

Work underlying parts of this dissertation has been previously published:

- Chapter 2, Chapter 3:
  - N. E. Croitoru. High-probability mutation in basic genetic algorithms. *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE*, pages 301–305, September 2014
- Chapter 4:
  - N. E. Croitoru. High probability mutation and error thresholds in genetic algorithms. *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE*, pages 271–276, September 2015
- Chapter 5:
  - N. E. Croitoru. Modelarea retelelor sociale folosind algoritmi genetici. Master's thesis, "Al. I. Cuza" University of Iasi, Romania, 2010
  - N. E. Croitoru. Lowering evolved artificial neural network overfitting through high-probability mutation. *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, September 2016

# 2. Initial Study

The focus on High-probability Mutation was initially motivated by the results of a preliminary study.

The goal of the study was to better choose and fine-tune probabilities for the Mutation and Crossover operators. A better choice for operator probabilities improves the quality of the optima found by the Genetic Algorithm - and this optimum quality is the measure for the success for a certain operator probability.

The design of the experimental phase is simple: Mutation and Crossover probabilities are varied from 0 to 1 in 0.01 increments - each percent being tested. For each Mutation probability and Crossover probability combination, a Genetic Algorithm instance is run to sample it 32 times, with the produced optima being recorded.

## 2.1. Explanation of choices

The full range of probabilities - opposite to e.g. investigating probabilities from 0 to 0.3 - especially for Mutation, was investigated due to a combination of factors, both intended and incidental.

Computational resource availability was favourable at the start of the experiment, as a combination of fast and well-understood algorithm implementation, time available for running the experiment, and simple CPU throughput. Additionally, a good parallel and remote algorithm execution infrastructure ensured minimal data loss and good resume capability after outages.

I thought, at the time, that having one less assumption (that good GA optima are found only for low-probability Mutation) would better suit a preliminary study, aimed at guiding further research. This choice was not made in the hope of finding good results for high-probability Mutation - at the time, I did expect to find the best results for low-probability mutation; it was merely a removal of an unnecessary assumption during an experiment design verification pass. Thus, the investigation into high-probability Mutation was not guided by inspiration or intuition, but by experiment design considerations.

## 2.2. Experiment Definition

In this experiment, the quality of the optima found by the GA are assumed to be a function of two parameters (mutation probability and crossover probability). Due to the heuristic nature of GAs, randomness is also expected to influence the results - this is controlled for by running multiple GA instances with the same parameters, and performing a random statistical sampling.

The experiment design is simple: mutation probability $p_m$ and crossover probability $p_{cx}$ are varied from 0 to 1 in 0.01 steps. Each $(p_m, p_{cx})$ pair is sampled 32 times, by running a GA and recording the optimum it proposes each time.

The Genetic Algorithm itself has the following features:

- Encoding: for each number representing a parameter for a numeric evaluation function, a separate part of a binary-encoded genome is allocated. Thus, each part of the solution representation is binary-encoded and separate.
- Representation precision: each number stored on binary-encoded genome is allotted enough bits to guarantee a precision of $10^{-5}$.
- Mutation: position independent, constant-probability, in-place bit-flip.
- Crossover: single-point, position independent, constant-probability. Crossover is

also non-destructive: crossover 'offsprings' are appended to the population, and do no replace their 'parents'.

- Selection: fitness-proportionate selection ('Roulette-wheel'). It also serves to regulate population size, after crossover offsprings are added - it always selects for a limited number of genomes, equal to the population size.

- Population: the number of genomes in a generation is constant (except after crossover) and set to 100.

- Stop condition: a soft stop-condition is used. The GA runs for 1000 generations; if a better solution was found in the past 100 generations, the run is extended for another 100 generations, after which the same evaluation is performed. The number of generations a GA runs for depends on problem instance; in practice, 1000 to 1300 generation-long runs are not uncommon.

- Initialisation: each bit in each genome in the population is randomly initialised either 0 or 1, with a 0.5 probability.

- Random numbers: a Mersenne19937[25] random number generator, initialised from system time and process ID (itself based on system memory state) generates each random choice. The C++ BOOST[1] Mersenne19937 implementation was used.

- Optimum reporting: While the GA does not preserve the best solutions within the population (non-elitism), the best candidate solution in each generation is registered; at the end of the algorithmic run, the best registered solution is selected as the optimum candidate found by the GA in that run.

- Structure: the Genetic Algorithm general structure follows this pseudocode scheme:

```
initialisation()
do:
    mutation()
    crossover()
    selection()
until(stopCondition)
```

The test problems used are Rosenbrock's Function[38] and the Six-Hump Camel Back Function[14], both in their 2-dimensional instances. For both functions, the optimum is a minimum.

The number of individual GA runs is $2 \times 101 \times 101 \times 32$: the number of function instances times the number of $p_m$ values chosen, times the number of $p_{cx}$ values, times the sample size for each parameter set.

## 2.3. Results

The results are presented in bi-dimensional plots. By taking all the results into account, but ignoring, in turn, $p_m$ and $p_{cx}$ values associated with them, 2D plots of optimum vs. $p_{cx}$ and, respectively, optimum vs. $p_m$ can be produced.

The box-plots in this section display, for each sample, the median, and the first and third quartiles. The "whiskers" extend up to $1.5 \times IQR$.

### 2.3.1. Crossover

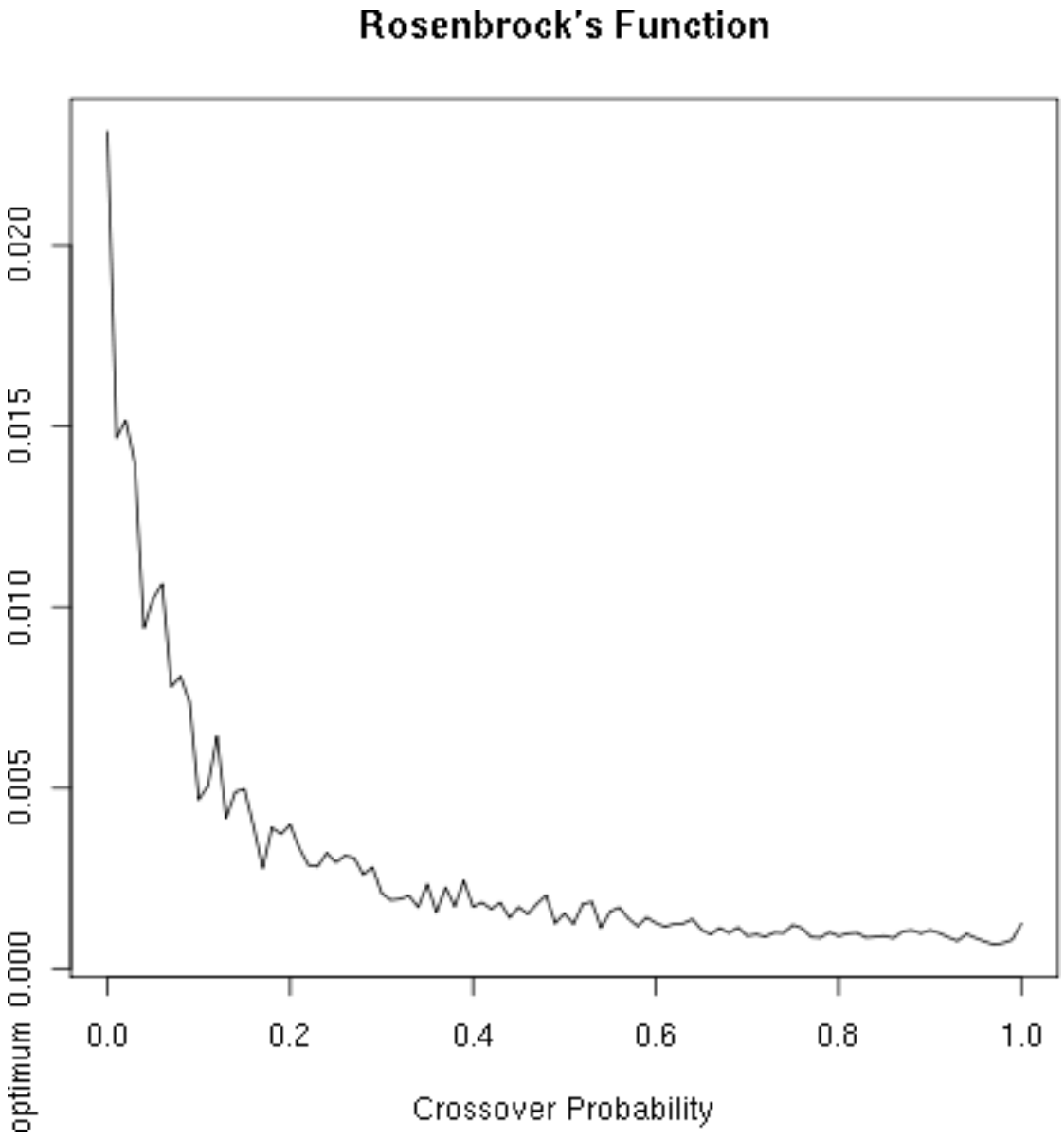


Figure 2.1.: Rosenbrock's Function: $p_{cx}$ vs. optimum quality, median

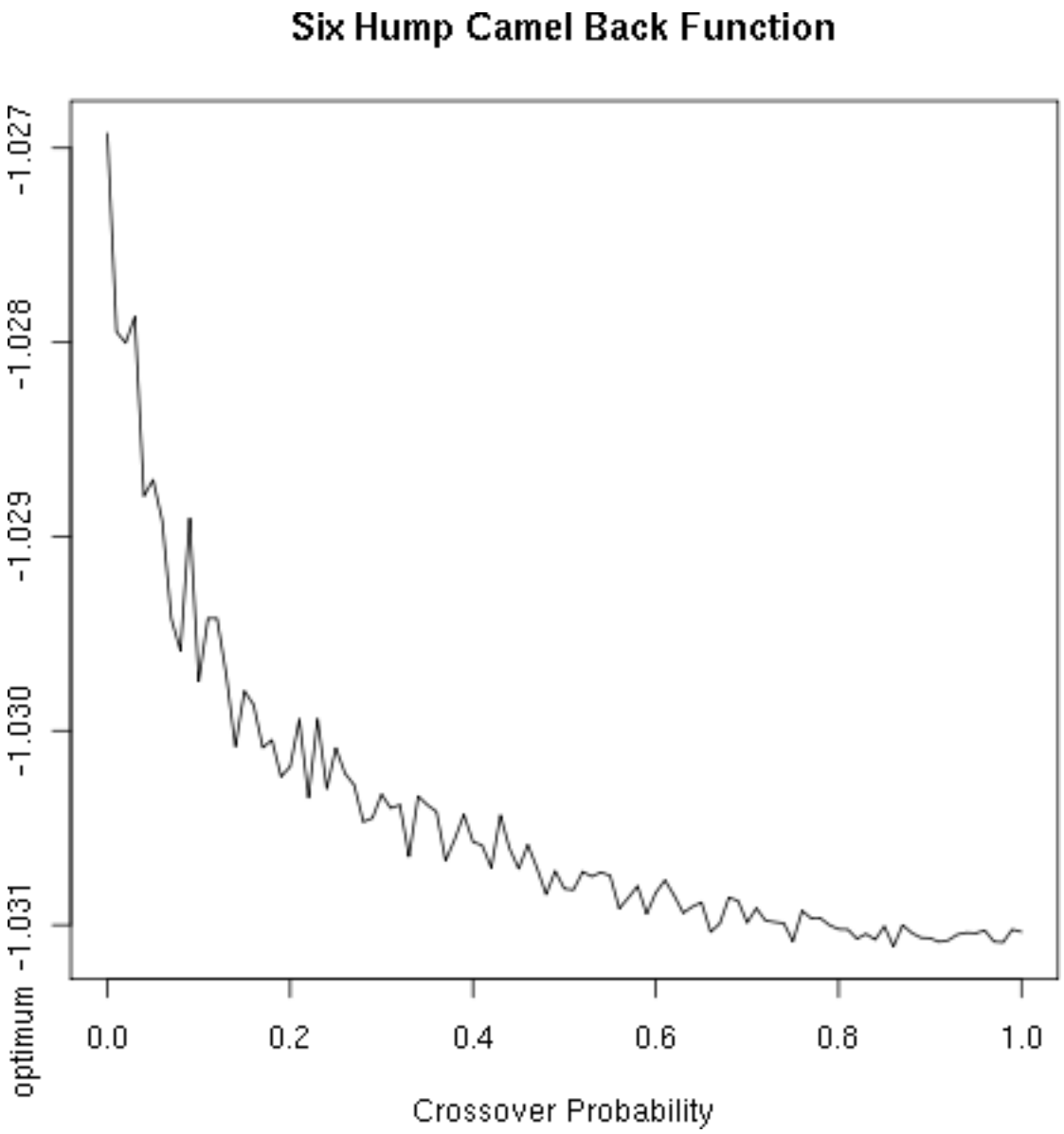


Figure 2.2.: Six-Hump Camelback Function: $p_{cx}$ vs. optimum quality, median

Both function instances show a general improvement in optimum quality as $p_{cx}$ improves. While, for Rosenbrock's Function (Fig. 2.1), the improvements plateau somewhat, the Six-Hump Camel Back Function (Fig. 2.2 takes advantage of every increase in $p_{cx}$; this means that $p_{cx}$ increases can be neutral with respect to found optima. Combined with the computational cost of non-destructive crossover, it shows that choosing a non-1

crossover probability is a valid strategy, with no prescribed answer.

The more detailed view into the statistical distribution of optima provided in Fig. 2.3 and 2.4 does not contradict the above conclusions.

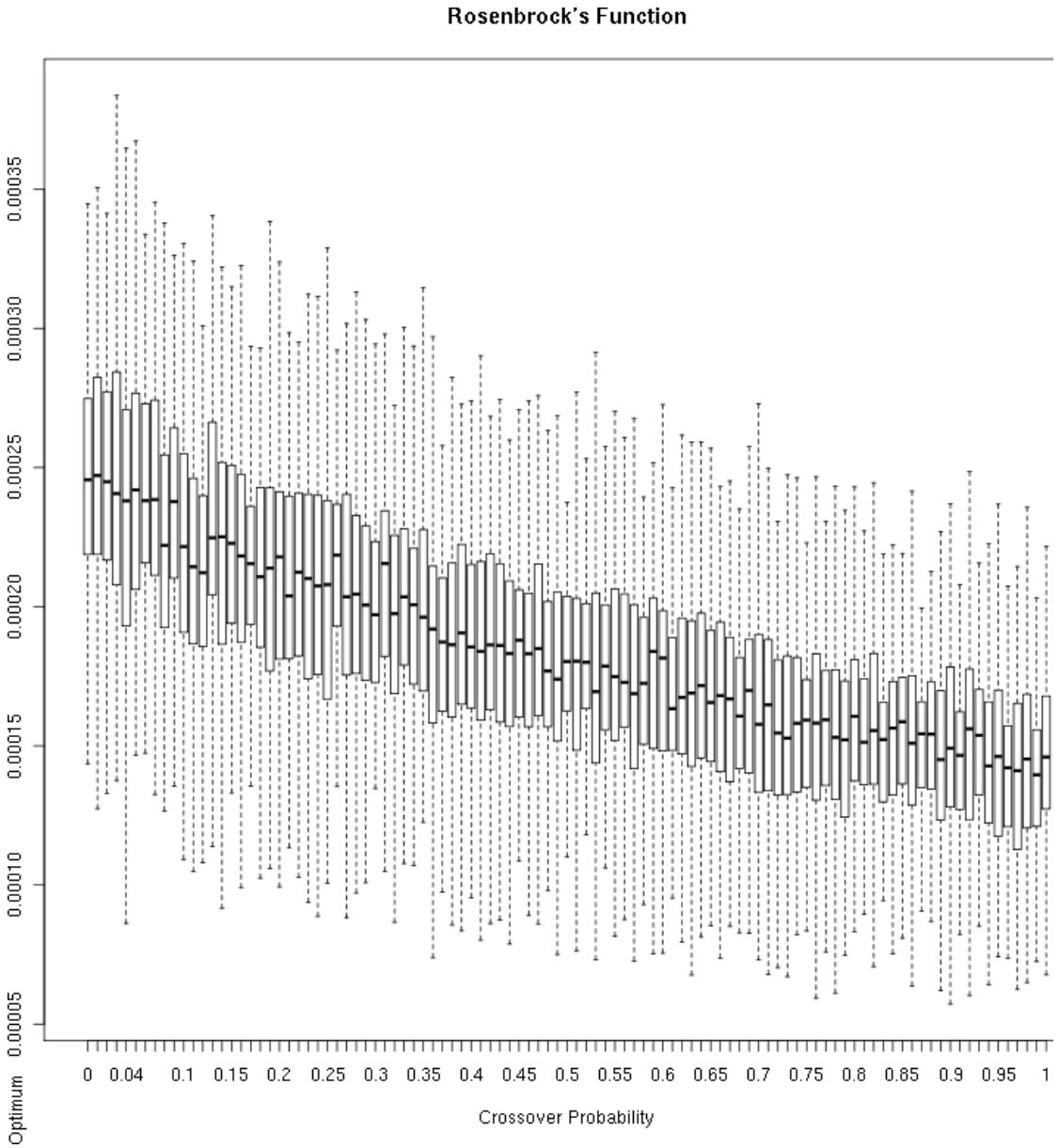


Figure 2.3.: Rosenbrock's Function: $p_{cx}$ vs. optimum quality, boxplot

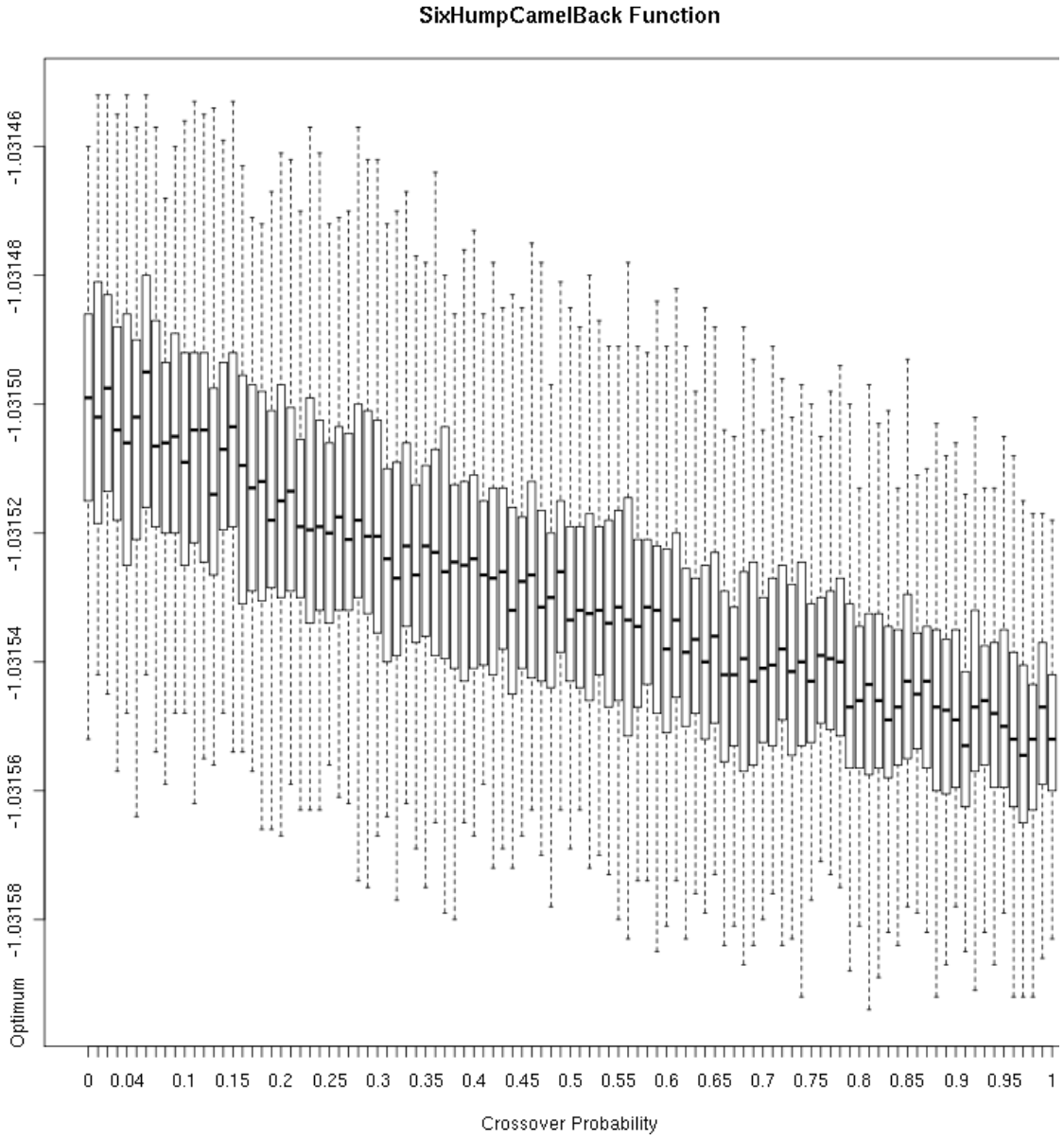


Figure 2.4.: Six-Hump Camelback Function: $p_{cx}$ vs. optimum quality, boxplot

### 2.3.2. Mutation

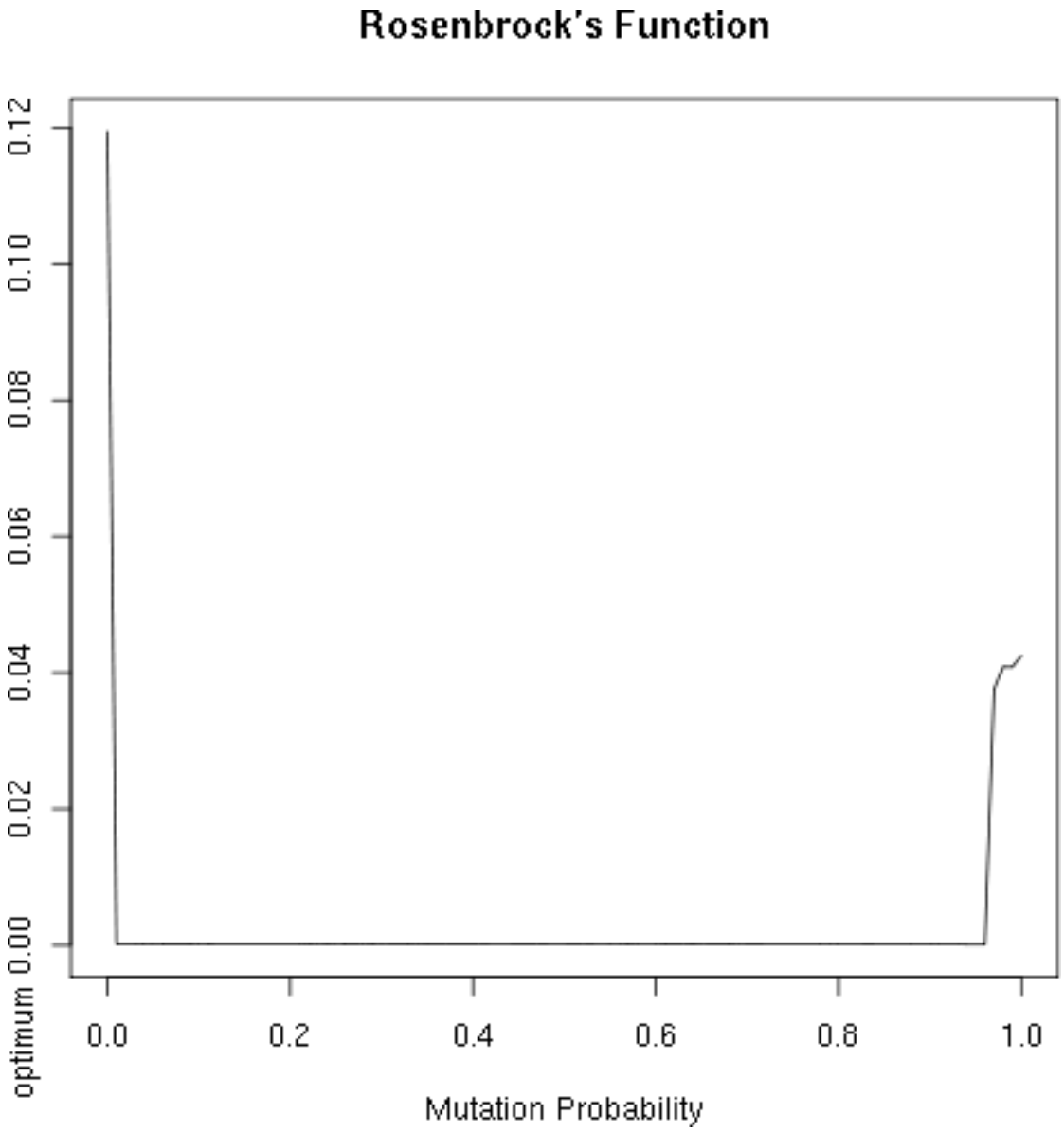


Figure 2.5.: Rosenbrock's Function: $p_m$ vs. optimum quality, median

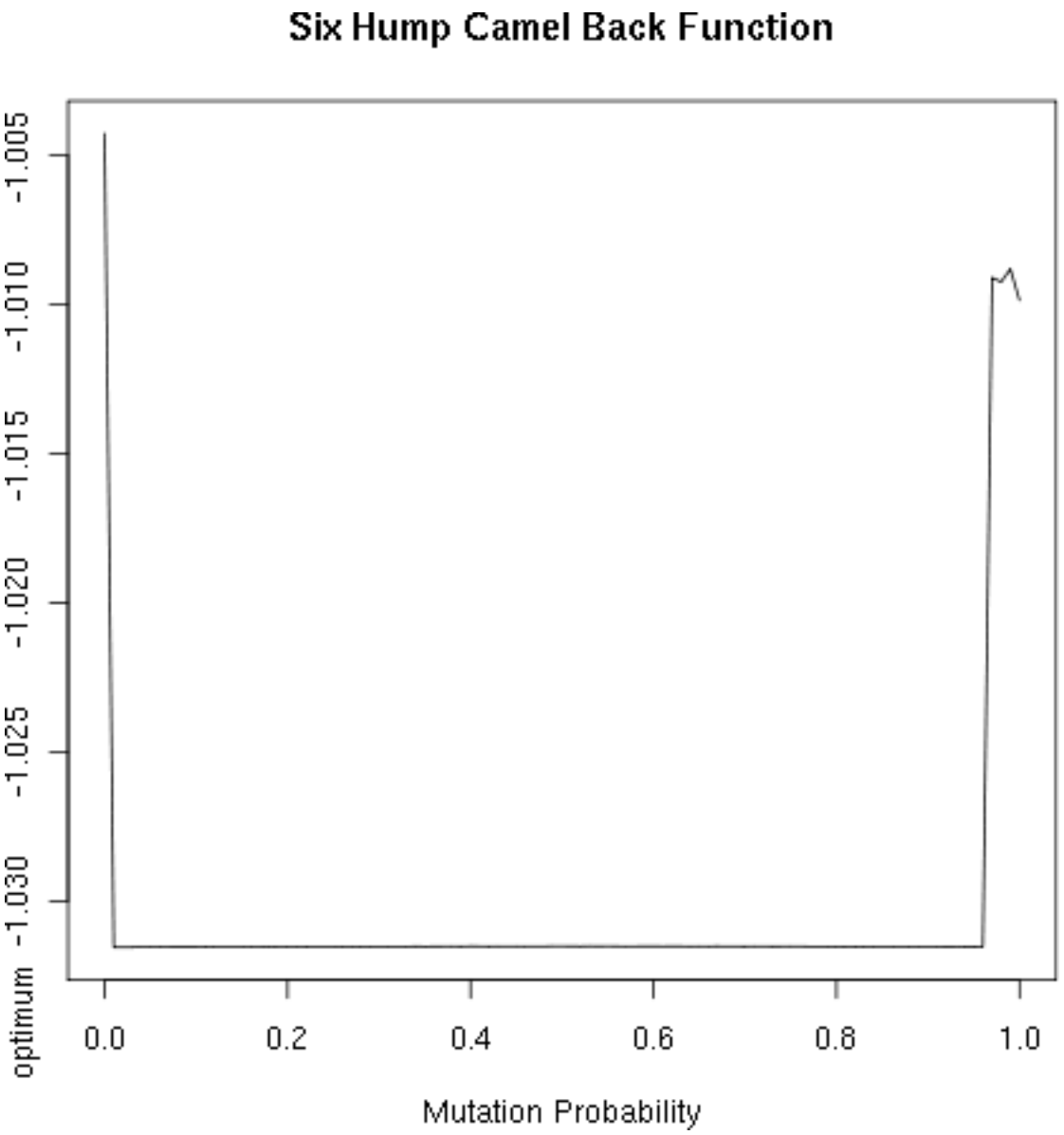


Figure 2.6.: Six-Hump Camelback Function: $p_m$ vs. optimum quality, median

Analysing the whole range of mutation probabilities provides little detail (Fig. 2.5 and 2.6). In the case of both function instances, no mutation at all proves highly detrimental to the GA. This can be explained through a lack of exploration in the GA (except that provided through the random initialisation step) - the algorithm will keep exploiting the same small regions of the search space through crossover, and significant solution

improvement (for which exploration is needed) is highly unlikely. Near-1 mutation also proves detrimental.

Having drawn conclusions pertaining to extreme values for $p_m$, we can crop those values out, and analise the rest of the data (while still maintaining a linear view, as opposite to a logarithmic one). This is done by entirely removing the optima corresponding to $p_m \in \{0, 0.97, 0.98, 0.99, 1\}$. The influence of those $p_m$ values can be seen in the plots above (Fig. 2.5 and 2.6), while the plots below (Fig. 2.7 and 2.8) show the influence of the other sampled $p_m$ values - both plots (for each function instance) assemble to offer a complete picture of all the $p_m$ values sampled.

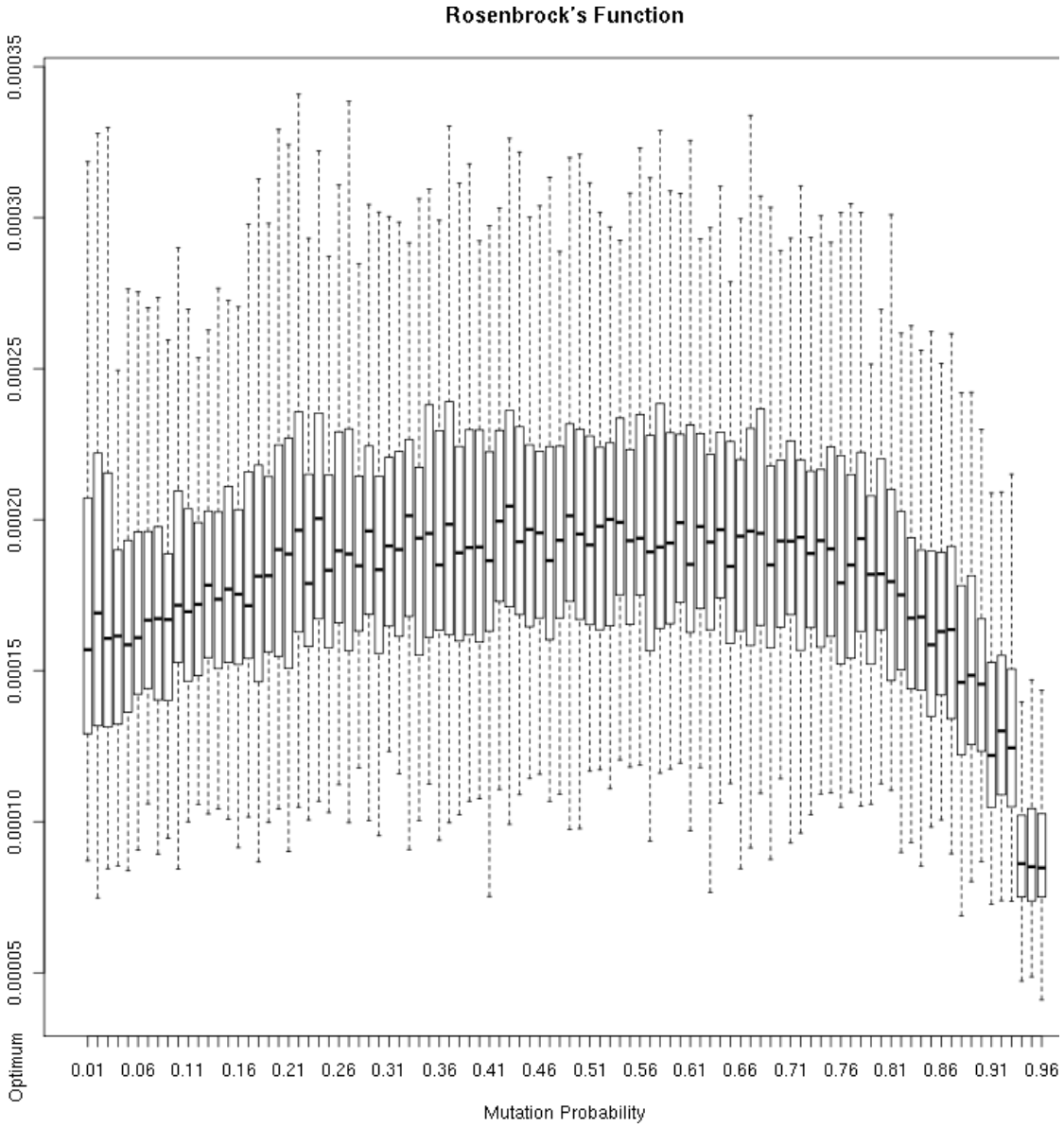


Figure 2.7.: Rosenbrock's Function: $p_m$ vs. optimum quality, boxplot

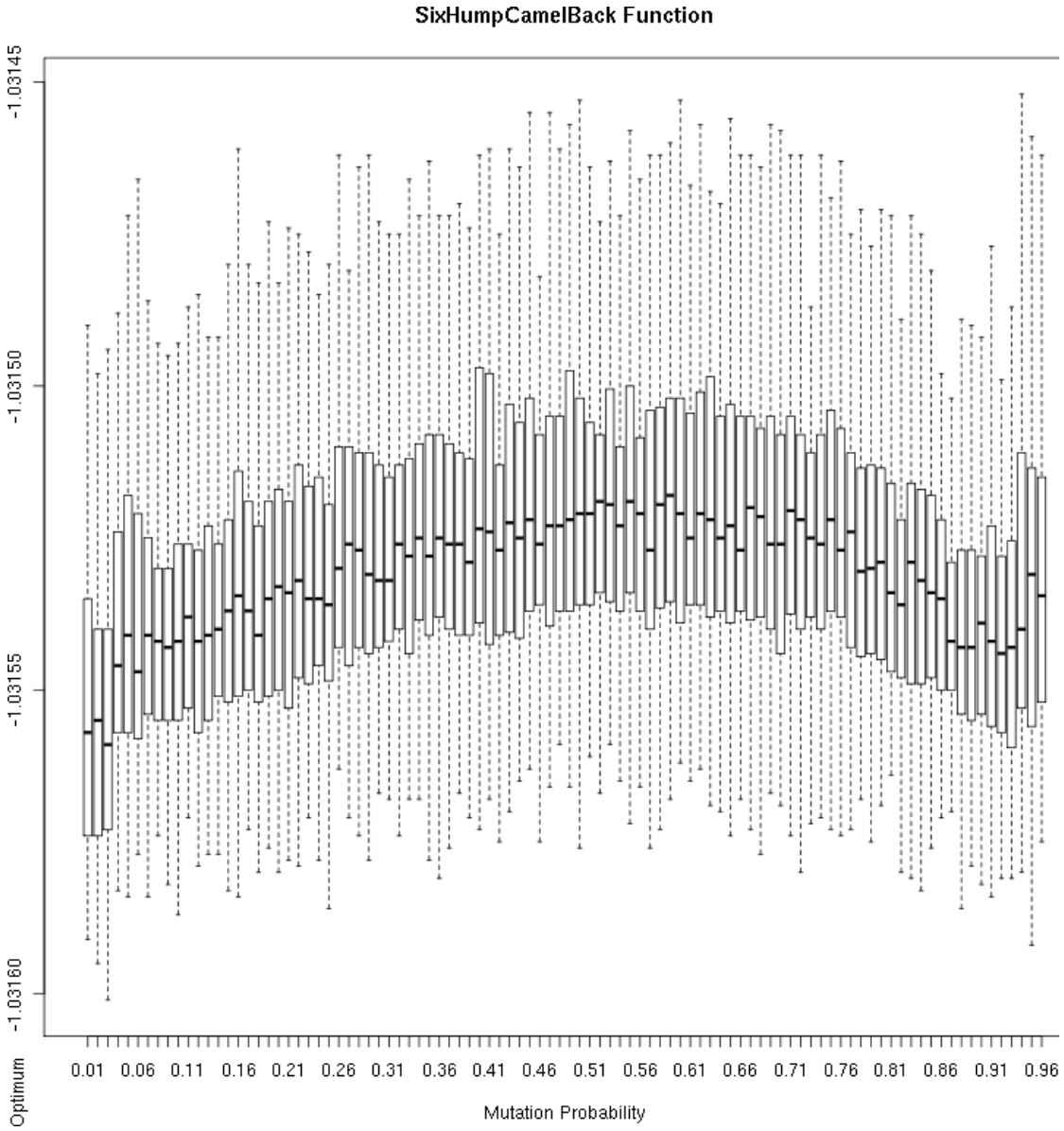


Figure 2.8.: Six-Hump Camelback Function: $p_m$ vs. optimum quality, boxplot

The plots (Fig. 2.7 and 2.8) show a decrease in optimum quality as $p_m$ is increased away from 0 [1]. However, this deterioration plateaus for $p_m$ values around $(0.5, 0.6)$. Afterwards, the optimum quality increases as $p_m$ increases.

[1] Since the optimum is a minimum, a decrease in optimum quality means an increase in numeric value for the proposed solutions

It is this good optimum quality at high $p_m$ values that motivated the later studies. Specifically, in the case of Rosenbrock's Function, the optima found for $p_m \in \{0.94, 0.95, 0.96\}$ are the best among all tested mutation probability values[2]. In the case of the Six-Hump Camelback Function, even though high-probability mutation performs worse than low-probability mutation, it still delivers good performance (above-average, among the mutation probability values tested); this too was interpreted as another reason to investigate High-Probability Mutation.

## 2.4. Interpretation

The impact of Crossover on GA performance in this experiment can be described as a relatively simple relationship, at least in rough detail: more Crossover translates, in general, as better found optima.

The local variations from this rule are small (Fig. 2.1 and 2.2). Very low $p_{cx}$ means low optimum quality, higher $p_{cx}$ means higher optimum quality. The range of variation is $\approx 0.025$ for Rosenbrock's Function (Fig. 2.1) and 0.004 for the Six-Hump Camelback Function (Fig. 2.2).

The influence of mutation probability is more complex, further away from simple linearity. No mutation at all, or $p_m = 1$ both show very poor results. In the first case, the GA cannot access entirely new information, and is left with attempting to recombine old genetic information. If, during initialisation, random chance doesn't create the required genes for a good solution, it simply cannot be found. In the case of $p_m = 1$, the situation is similar: while all the bits are flipped back and forth in a synchronised manner, new information is not brought to the GA. The slight improvement over $p_m = 0$ can be attributed to the GA having twice as many variants to exploit.

Less extreme mutation probability values show the best results in close proximity to

[2] A visual indication can be found in Fig. 2.7, where samples with $p_m \in \{0.94, 0.95, 0.96\}$ show interquartile Q1-Q3 separation from all other samples

0 and 1, while the worst (except those already discussed) optima values are found in a wide range around $p_m = 0.5$ (Fig. 2.5 and 2.6).

I interpret these results in the following way: mutation probabilities close to 0 do not scramble the information contained in the genomes, allowing evolution (in this case, new genomes being very similar to previous ones) to occur.

As $p_m$ increases to 0.5 and beyond, genomes are scrambled until evolution is disrupted (at least partially), and Random Search appears. At $p_m = 0.5$, each gene has a 0.5 chance to be 0 and a 0.5 chance to be 1 (immediately after mutation). The value at a certain locus no longer depends on previous values - on the steady evolution during past generations, within the Genetic Algorithm - but on the random chance at the current mutation step. Crossover merely recombines uniformly random genes in an uniformly random way; selection does skew the population distribution towards better fitness, but there can be little persistence, since the population is uniformly scrambled at the next mutation step.

However, as $p_m$ increases even further, genes have a greater chance to be changed back to where they started. Information across generations is again preserved, and evolution can take place. For e.g. $p_m = 0.9$, the value of at a certain locus will be anti-correlated with the previous value at that position; after two generations, the anti-correlation will result, for the binary encoding, in a likely return to the initial value. Since these changes are synchronised across the whole population, all the bits will be flipped in step, and the genetic information will 'reappear' in its standard encoding every other generation.

High-Probability Mutation is not a Random Search Algorithm, but a basic way for the Genetic Algorithm to work with two encodings - a base encoding and its binary negation.

# 3. High-probability Mutation

## 3.1. Description

In the context of Genetic Algorithms, high-probability mutation is defined by two competing measurements:

- the basic mutation probability - the simple average rate of applying mutation to any locus - is high;
- the effective mutation probability - the cumulated rate of gene change brought upon by mutation in the course of the algorithm - is low.

For binary-encoded Genetic Algorithms, the effective mutation rate can be measured every two generations. This is due to the fact that any locus has only two alleles, and because basic mutation simply flips between the two possible values of a locus. After 1 generation, a high-probability mutation may have flipped most of the bits in the population, but after 2 generations, the same high-probability mutation will have flipped most bits back.

The effective mutation rate after 2 generations for binary-encoded genomes can be calculated by considering the outcomes of each possible mutation. The reasoning below works for constant mutation rates, and can be extended to cover small variations in mutation rate (genome-dependent, locus-dependent or time-dependent).

Assuming a random locus, there are two outcomes as the mutation operator is applied in a generation: either the bit is flipped, or it is not. Let us name these two events $f$,

for flipped, and $\overline{f}$ for not flipped. Obviously, they are statistically complementary, the chance of $f$ occurring is $p_f = 1 - p_{\overline{f}}$.

During the second mutation pass (in the following generation), the same locus has the same chances of being flipped or not. For the information contained at that locus, there are two possibilities: it has either retained the value it has started two generations with, or it has not.

The probability of a bit-flip is equal to the mutation probability, $p_f = p_m$ (in the constant mutation probability case) . However, the rate of effective mutation, $p_{em}$, represents the rate of information change over the course of two generations. The relation between the $p_m$ and $p_{em}$ can be seen by following the chain of events that lead to each outcome.

- For a locus to have changed its value, there are two possible scenarios:
  - It had been changed by the first mutation, and was unchanged by the second.
  - It was changed by the second mutation pass, but remained unchanged during the first.
- For a locus to have preserved its value, there are two possible scenarios:
  - It has remained unchanged during both mutation passes.
  - It has been bit-flipped by both mutation passes.

Thus, the probability for a change (over the course of two successive generations is):

$$p_{em} = (1 - p_m) \cdot p_m + p_m \cdot (1 - p_m) = \tag{3.1}$$

$$= 2 \cdot p_m \cdot (1 - p_m) \tag{3.2}$$

Its opposite, the probability for retaining the initial value, is:

$$p_{\overline{em}} = (1 - p_m)^2 + p_m^2 = \tag{3.3}$$

$$= 1 - 2 \cdot p_m + 2 \cdot p_m^2 = \tag{3.4}$$

$$= 1 - 2 \cdot p_m \cdot (1 - p_m) \tag{3.5}$$

which verifies the relationship $1 - p_{em} = p_{\overline{em}}$.

The $p_{em} = 2 \cdot p_m \cdot (1 - p_m)$ expression is a quadratic function of a single parameter ($p_m$). It has two solutions, $p_m = \{0, 1\}$, and a stationary point at $p_m = 0.5$. Since $p_m$ is a probability, the function domain is $[0, 1]$.

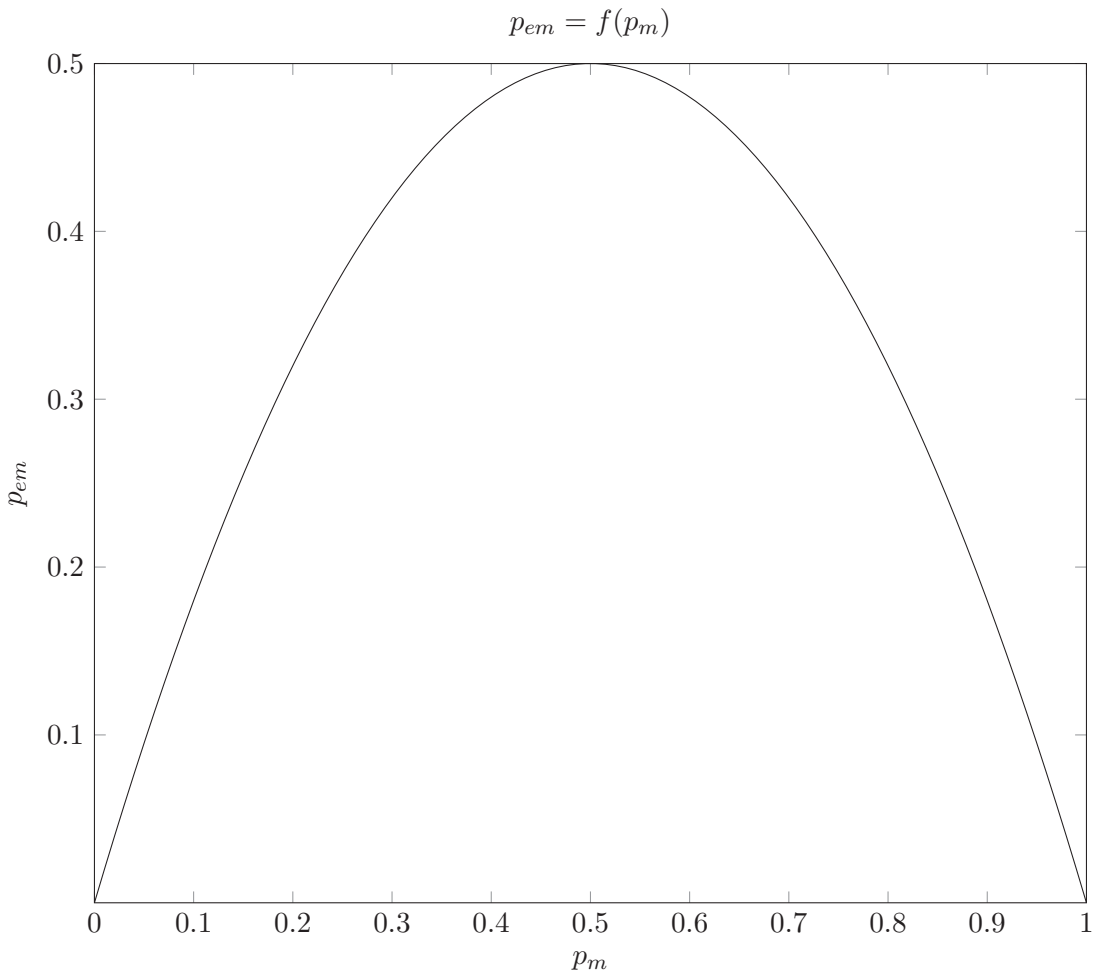


Figure 3.1.: Effective mutation probability as a function of mutation probability.

The maximum of $p_{em} = 0.5$ is reached at $p_m = 0.5$. The minima of $p_{em} = 0$ are reached for $p_m \in \{0, 1\}$.

There are similarities between the above plot (Fig. 3.1) and the relation between optimum quality and mutation probability for Rosenbrock's Function (Fig. 2.5) and the Six-Hump Camelback Function (Fig. 2.6). A low effective mutation rate corresponds with good optima, and high $p_{em}$ with bad optima - except for the very extremities of the probability interval. Effective mutation rates are equal for complementary mutation probability: $p_{em}(p_m) = p_{em}(\overline{p_m})$. The effective mutation rate over two generations for a

$p_m = 0.95$ is $2 \cdot 0.95 \cdot (1 - 0.95) = 2 \cdot 0.95 \cdot 0.05 = 0.095$.

In this context, $p_{em}$ is a better predictor for GA solution quality than $p_m$. Low-probability mutation explores one region of promising $p_{em}$ values, while high-probability mutation explores the other region.

## 3.2. Performance Comparison

Choosing between low-probability mutation and high-probability mutation remains an open question; as Chapter 2 shows, for different function instances we obtain different performances. Furthermore, mutation interacts with the rest of the Genetic Algorithm: hypotheses formulated only in terms of mutation need to be tested in the context of the whole GA. Even the simple $p_{em}$ formula does not hold accurately since, after each generation, selection skews the probability distribution of genes according to fitness landscape.

In order to help choose between probabilities and understand mutation better, a wider set of function instances has been investigated. These functions are selected from well-known benchmark numerical functions. Since high-probability mutation is not limited to binary representations of integer or fixed-point numbers, and since low-probability Simple Genetic Algorithms have difficulty solving them, two bit-block functions have been tested: GA Royal Road[28] and the GA Trap Function[13].

The aims of the experiment are to better understand high-probability mutation interaction with fitness landscapes (by comparing GA behaviour on different function), to detect high-probability mutation problems on bit-block functions, and see if it can improve GA performance on functions that traditionally favour other algorithms.

The experimental set-up is as follows:

- 3 low-probability mutation values and 1 high-probability mutation values are used: $p_m \in \{0.0005, 0.001, 0.01, 0.95\}$.

- 2 crossover probability values are used: $p_{cx} \in \{0.2, 0.5\}$.
- The numeric function instances used are: De Jong 1[12], Michalewicz's [26], Rastrigin's[37], Rosenbrock's [38], Schwefel's[40] and Six-Hump Camelback [14], each 2-dimensional, with a guaranteed $10^{-5}$ precision.
- The bit-block functions used are: GA Royal Road[28] and GA Trap Function[13], each having 4 blocks of 8 bits.
- Each parameter set (mutation probability, crossover probability and function instance) is sampled 1024 times.

The GA set-up is similar to that described in Section 2.2; the changes are in $p_m$ and $p_{cx}$ values sampled and in setting a population size of 50.

The reduction in $p_m$ and $p_{cx}$ values investigated, alongside a population size reduction (which cuts the number of evaluations during a GA run in half) allows for a greater number of statistical samples to be gathered. This increase translates to a grater statistical significance of comparisons performed.

### 3.2.1. Function instance definition

To better understand Mutation and GA behaviour, the numeric function instances used are described below. For the surface plots, only grid intersections have been computed, while the grid itself and the plot surface have been interpolated.

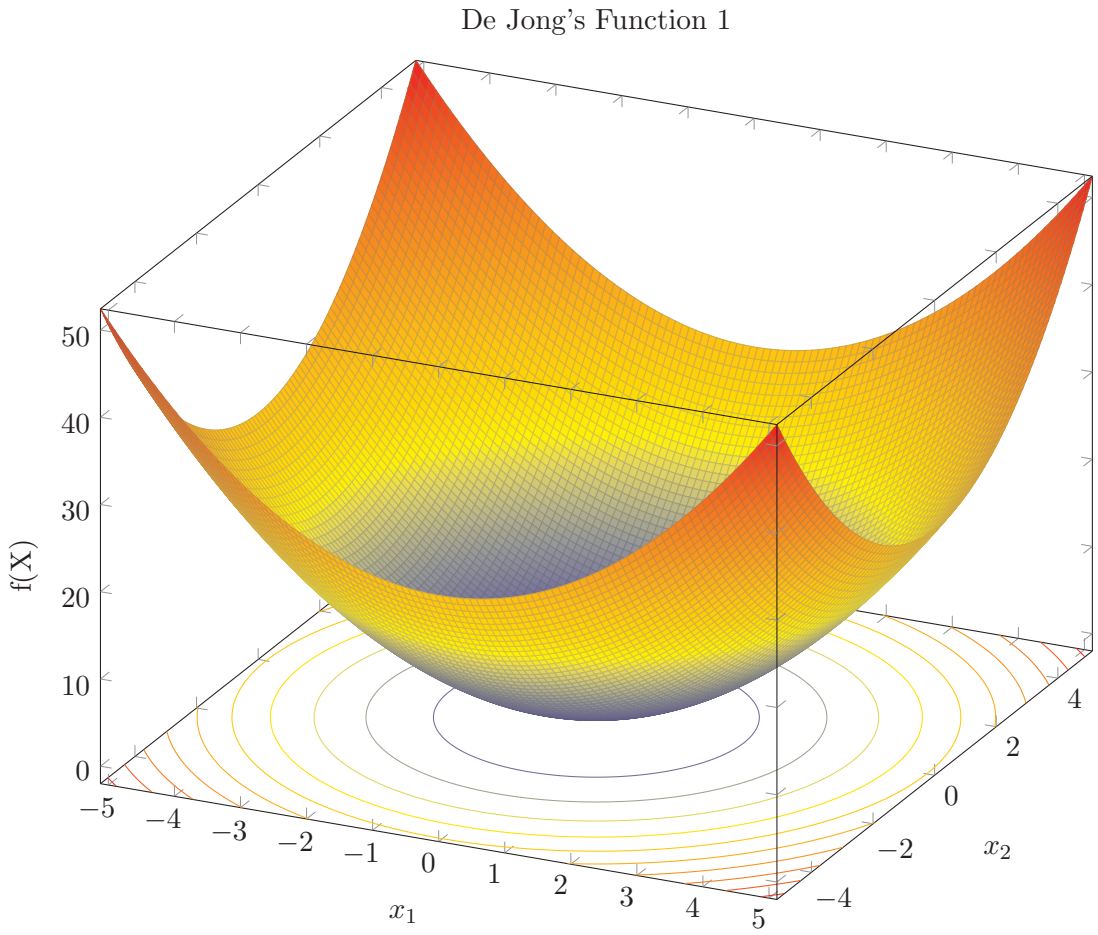


Figure 3.2.: De Jong's Function 1, 2-dimensional plot

De Jong's Function 1: $f(X) = \sum_{i=1}^{n} x_i^2, X = \{x_1, \ldots, x_n\}, -5.12 \leq x_i \leq 5.12$

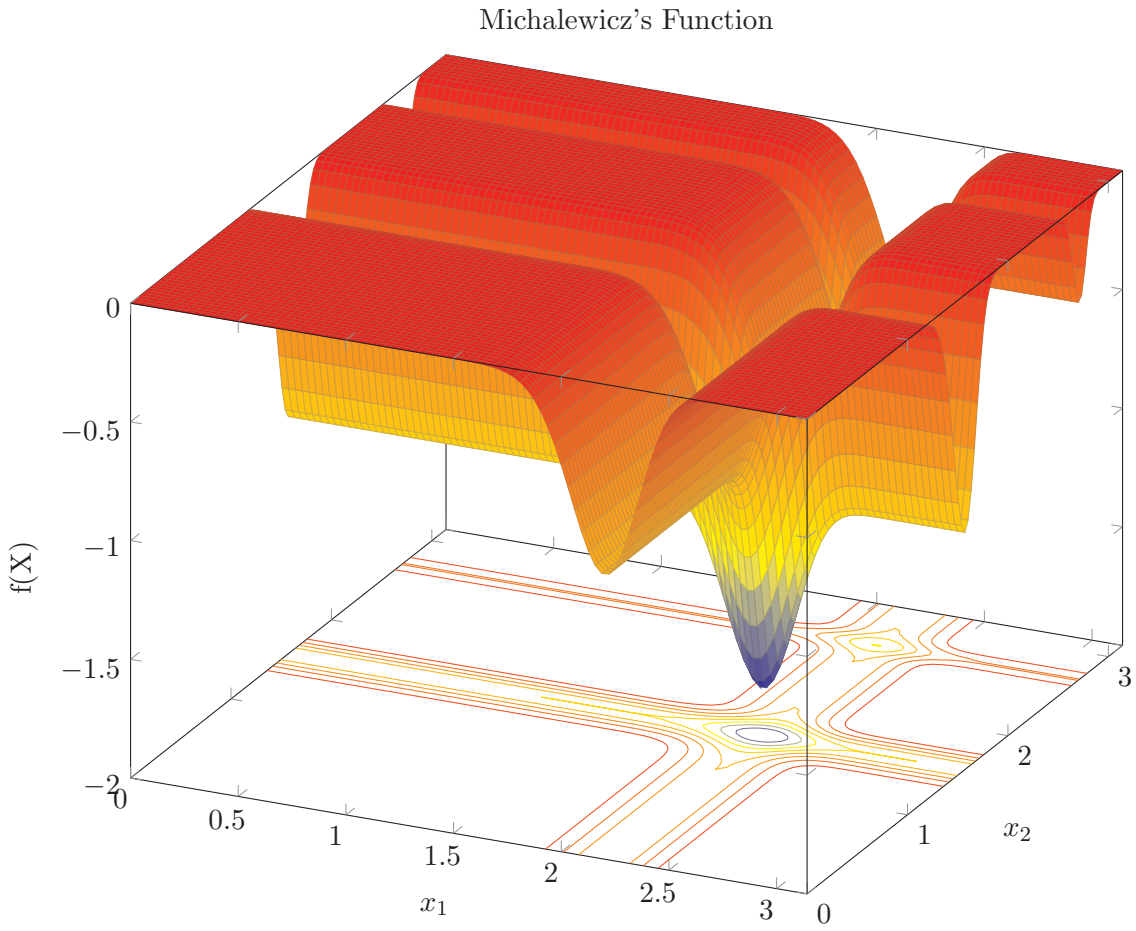


Figure 3.3.: Michalewicz's Function, 2-dimensional plot

Michalewicz's Function: $f(X) = -\sum_{i=1}^{n} sin(x_i) \cdot \left(sin\left(\frac{i \cdot x_i^2}{\pi}\right)\right)^{2 \cdot m}, X = \{x_1, \ldots, x_n\}$, $0 \leq x_i \leq \pi, m = 10$

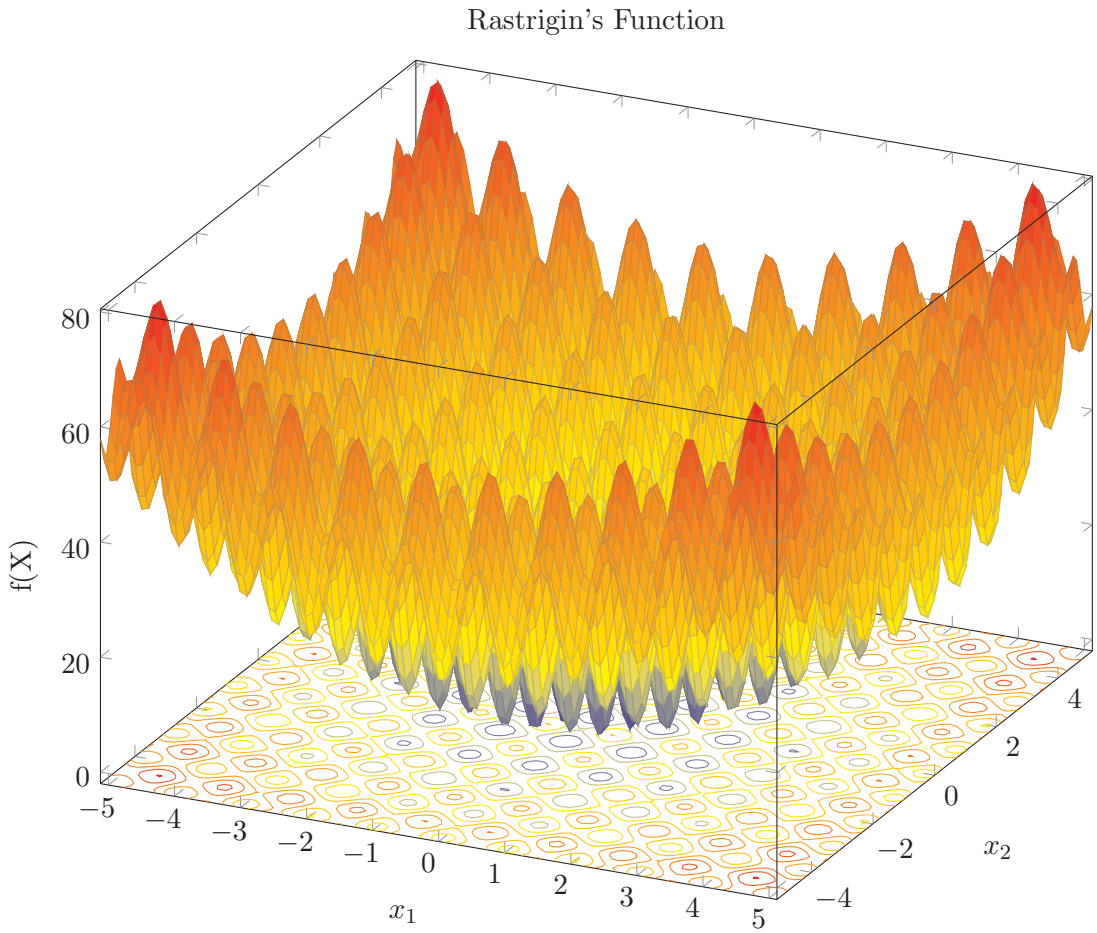


Figure 3.4.: Rastrigin's Function, 2-dimensional plot

Rastrigin's Function: $f(X) = 10 \cdot n + \sum_{i=1}^{n} \left(x_i^2 - 10 \cdot cos(2 \cdot \pi \cdot x_i)\right), X = \{x_1, \ldots, x_n\}$, $-5.12 \leq x_i \leq 5.12$

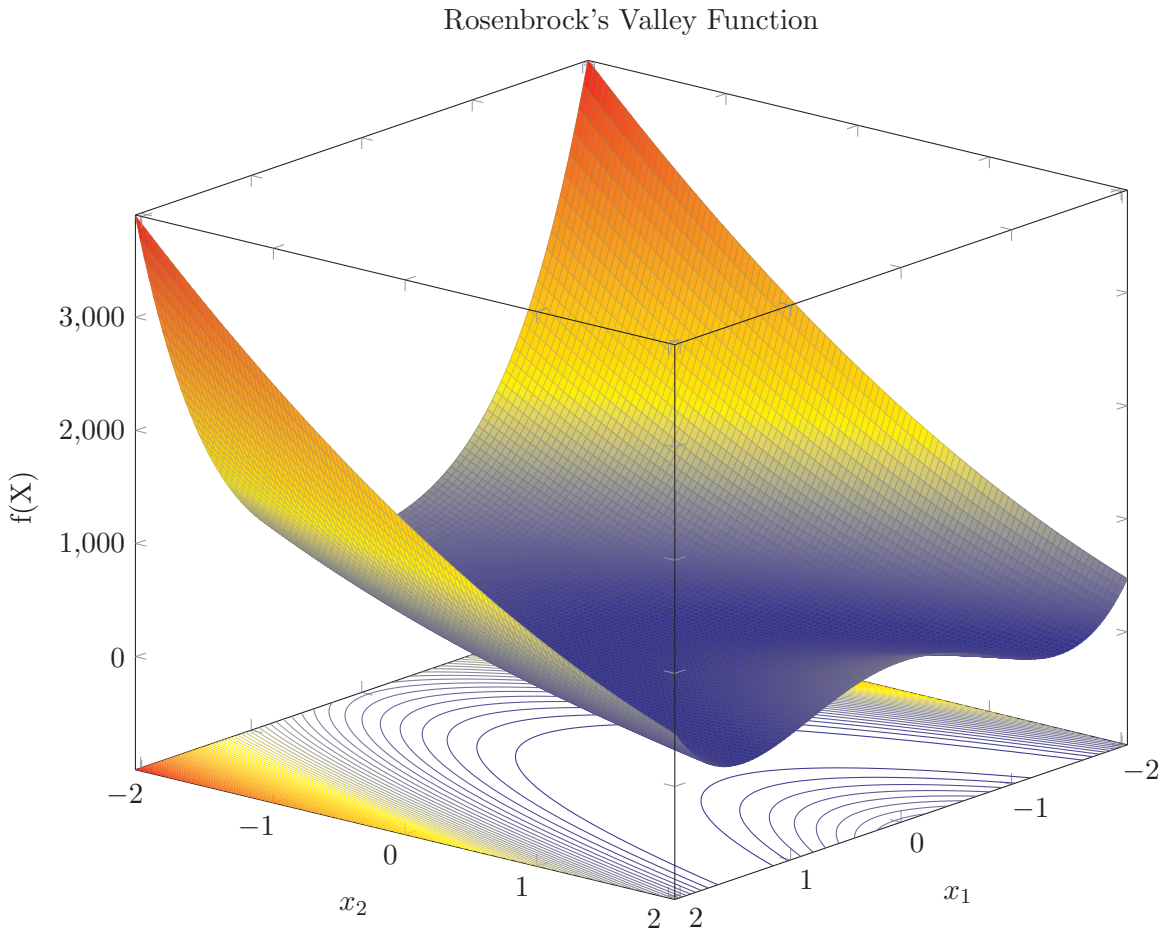


Figure 3.5.: Rosenbrock's Valley Function, 2-dimensional plot

Rosenbrock's Valley Function:

$$f(X) = \sum_{i=1}^{n-1} 100 \cdot \left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2, X = \{x_1, \ldots, x_n\}, -2.048 \leq x_i \leq 2.048$$

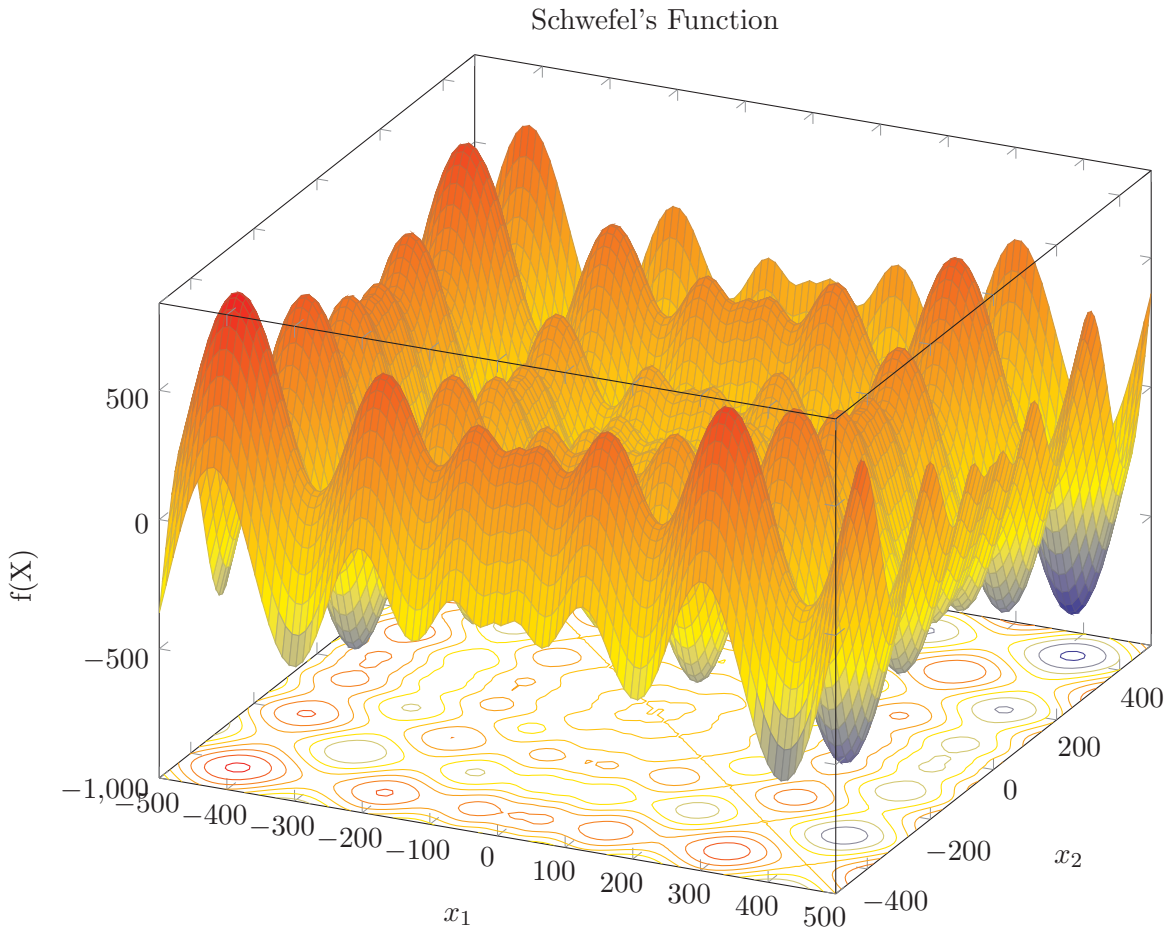


Figure 3.6.: Schwefel's Function, 2-dimensional plot

Schwefel's Function $f(X) = \sum_{i=1}^{n} -x_i \cdot sin(\sqrt{|x_i|}), X = \{x_1, \ldots, x_n\}, -500 \leq x_i \leq 500$

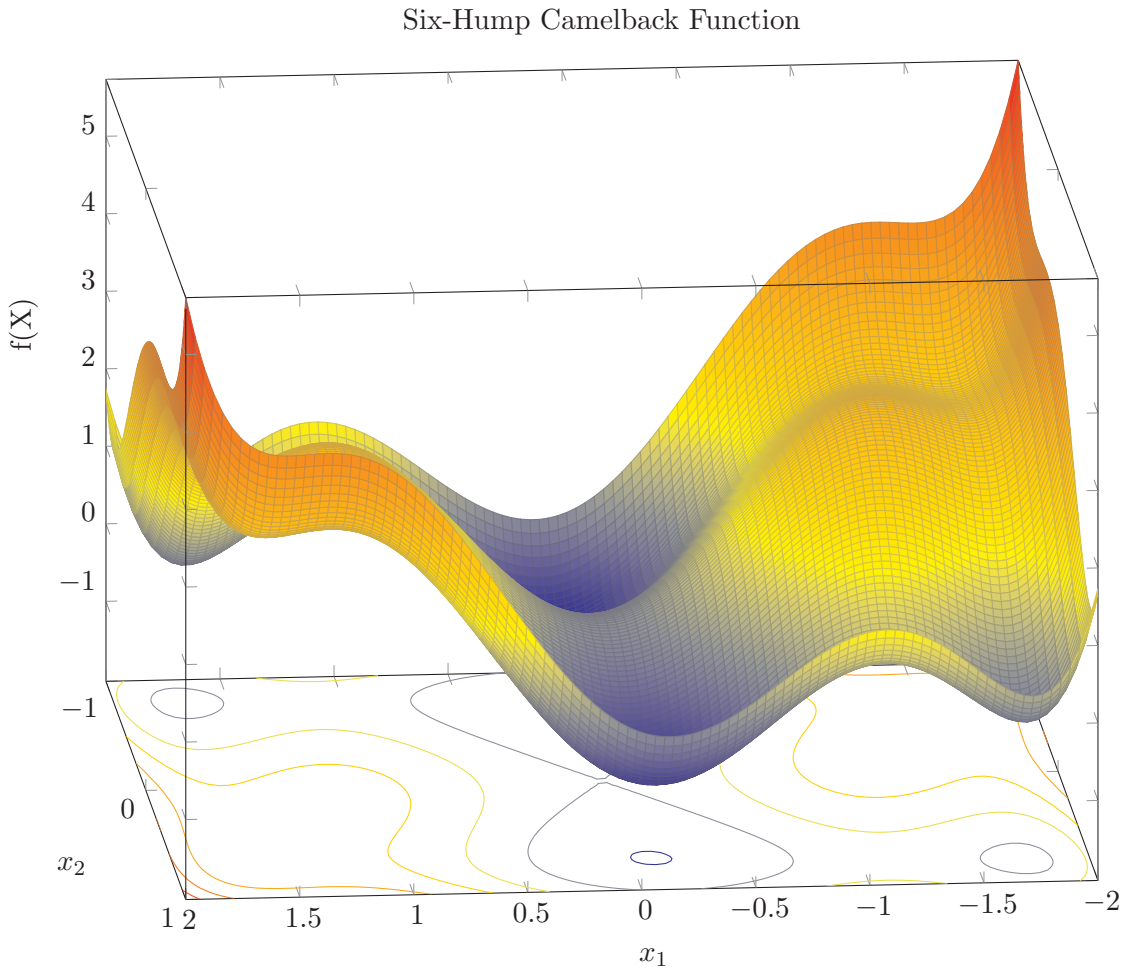


Figure 3.7.: Six-Hump Camelback Function, plot

Six-Hump Camelback Function: $f(X) = \left(4 - 2.1 \cdot x_1^2 + \frac{x_1^4}{3}\right) \cdot x_1^2 + x_1 \cdot x_2 + (x_2^2 - 1) \cdot 4 \cdot x_2^2$, $X = \{x_1, x_2\}$, $-3 \leq x_1 \leq 3, -2 \leq x_2 \leq 2$

### 3.2.2. Results

The results are divided in multiple tables, first by function type (numerical function or bit-block), then by crossover probability (first $p_{cx} = 0.2$, then 0.5). The numerical functions have minimal optima, while the bit-block functions have maximal optima. For better identification, an upwards arrow ↑ next to the function name signifies an optimum that is a maximum, while a downwards arrow ↓ a minimum.

Teach table cell contains 3 numerical values, placed vertically. Each displays different information about the statistical sample[1] collected for each parameter set. Starting with the top value, they signify: the median of the sample, the mean of the sample and the standard deviation of the sample:

| median |
| --- |
| mean |
| standard deviation |

Median values appearing in **bold font face** mark the best median for that function instance. Average values so marked mean that, following a statistical test, a significant difference has been observed between that sample and all others. Furthermore, it means that sample is better (closer to the optimum).

The statistical test was the following: the samples were compared against the other, using Welch's unequal variance t-test[43] ($\alpha = 0.01$). Welch's t-test was selected because the standard deviations among samples tend to differ. Afterwards, the resulting p-values were adjusted using the Holm-Bonferroni method[20]. The Holm method was chosen because it is more conservative[3], lowering the probability of false-positives (at the cost of increasing false-negative chance).

[1] Sample size is constant at 1024

| Function name | $p_{mutation}$ 0.0005 | 0.001 | 0.01 | 0.95 |
|---|---|---|---|---|
| De Jong 1 ↓ | 6.51e-05 | 6.81e-05 | **6.49e-05** | 6.31e-04 |
| | 0.00010 | 0.00010 | 0.00010 | 0.00009 |
| | 0.00012 | 0.00012 | 0.00011 | 0.00097 |
| Michalewicz↓ | **-1.80129** | -1.80128 | -1.80128 | -1.80062 |
| | -1.80127 | -1.80127 | -1.80126 | -1.80007 |
| | 3.48e-05 | 4.17e-05 | 4.44e-05 | 0.00169 |
| Rastrigin↓ | 0.00257 | **0.00243** | 0.00258 | 0.05708 |
| | 0.00634 | 0.00575 | 0.00622 | 0.10497 |
| | 0.01130 | 0.01023 | 0.01253 | 0.13632 |
| Rosenbrock↓ | 0.00027 | 0.00026 | 0.00027 | **9.91e-05** |
| | 0.00396 | 0.00538 | 0.00408 | **0.00006** |
| | 0.03789 | 0.04692 | 0.03394 | 0.00843 |
| Schwefel↓ | **-837.961** | -837.961 | -837.961 | -837.515 |
| | -831.379 | -831.199 | -829.648 | **-836.258** |
| | 25.49138 | 25.85371 | 28.13673 | 10.56529 |
| SixHump↓ | -1.03155 | -1.03156 | **-1.03156** | -1.03148 |
| | -1.03149 | -1.03150 | -1.031509 | -1.03137 |
| | 0.00015 | 0.00016 | 0.00014 | 0.00031 |

Table 3.1.: Low-probability vs. high-probability mutation. Numerical functions, $p_{cx} = 0.2$

Table 3.1: High-probability mutation is better than the low-probability mutation for only two functions: Rosenbrock's and Schwefel's. When it under-performs, high-probability mutation distances itself significantly and shows a comparatively large standard deviation. The best example to illustrate this is Rastrigin's Function.

When the investigated high-probability mutation rate outperforms the sampled low-probability mutation rates, the reverse tends to be true: samples obtained using high-probability mutation tend to show comparatively small standard deviations.

In the case of Schwefel's function, the median of the samples favours low-probability mutation. However, the mean, standard deviation and statistical test favour high-probability mutation.

| Function name | $p_{mutation}$ 0.0005 | 0.001 | 0.01 | 0.95 |
|---|---|---|---|---|
| De Jong 1 ↓ | 4.78e-05 | **4.44e-05** | 4.87e-05 | 0.00053 |
| | 8.38e-05 | **7.16e-05** | 8.15e-05 | 0.00077 |
| | 9.67e-05 | 8.08e-05 | 0.00010 | 0.00081 |
| Michalewicz↓ | -1.80129 | **-1.80129** | -1.80129 | -1.80080 |
| | -1.80127 | -1.80128 | -1.80128 | -1.800382 |
| | 3.10e-05 | 2.64e-05 | 2.91e-05 | 0.00123 |
| Rastrigin↓ | **0.00184** | 0.00210 | 0.00191 | 0.04979 |
| | 0.00482 | 0.00518 | 0.00417 | 0.08859 |
| | 0.00923 | 0.01031 | 0.00630 | 0.11634 |
| Rosenbrock↓ | 0.00020 | 0.00021 | 0.00019 | **8.40e-05** |
| | 0.00303 | 0.00334 | 0.00361 | **0.00077** |
| | 0.02902 | 0.03706 | 0.03021 | 0.02071 |
| Schwefel↓ | -837.962 | **-837.962** | -837.962 | -837.592 |
| | -831.511 | -832.3 | -831.899 | **-836.807** |
| | 25.15705 | 23.25951 | 23.89548 | 8.26985 |
| SixHump↓ | **-1.03158** | -1.03157 | -1.03157 | -1.03149 |
| | -1.03153 | -1.03153 | -1.03153 | -1.03141 |
| | 0.00014 | 0.00012 | 0.00010 | 0.00023 |

Table 3.2.: Low-probability vs. high-probability mutation. Numerical functions, $p_{cx} = 0.5$

Table 3.2: Increasing $p_{cx}$ to 0.5 produces largely similar results. For most function instances, results are improved, compared to $p_{cx} = 0.2$. In the case of Rosenbrock's Function, increasing the crossover probability slightly worsens the performance of high-probability mutation (in mean and standard deviation, but not in median).

| Function name | $p_{cross-over}$ 0.2 | | 0.5 | |
|---|---|---|---|---|
| | $p_{mutation}$ 0.01 | 0.95 | 0.01 | 0.95 |
| Royal Road$^{\uparrow}$ | 40 | 40 | 40 | **48** |
| | 42.5 | **47** | 41.625 | **48.125** |
| | 7.25 | 11.96 | 4.31 | 10.522 |
| Trap$^{\uparrow}$ | 53 | **56** | 53 | **56** |
| | 52.640 | **56.140** | 52.656 | **56.421** |
| | 1.2 | 1.48 | 1.15 | 1.29 |

Table 3.3.: Low-probability vs. high-probability mutation. Bit-block functions.

Table 3.3: High-probability mutation clearly outperforms low-probability mutation on the two bit-block function instances used. Increasing crossover slightly worsens low-probability mutation performance (and increases high-probability mutation performance) on the GA Royal Road Function, and generally lowers standard deviation.

### 3.2.3. Interpretation

Low-probability mutation and high-probability mutation show different relative performances on different function instances. It is interesting to note that low-probability mutation performs much better on function landscapes where the optimum is surrounded by relatively steep walls, e.g. Rastrigin's Function (Fig. 3.4). The simplest example to illustrate this is De Jong's Function 1 (Fig. 3.2), which consists of nothing but a single optimum surrounded by steep walls (it is continuous, strictly convex and multi-variate uni-modal). On these function landscapes, we find the largest relative difference between low- and high-probability mutation (approximately by a factor of 10 in median, mean and standard deviation, although these differences cover just a small part of the function's ranges).

By contrast, when a function landscape consists of relatively flat plateaus surrounding optima, high-probability mutation performs better (such as Rosenbrock's Function: Fig. 3.5). When the two types of features are mixed, the gap between low- and high-probability mutation shrinks.

I interpret these results as an increase in exploration on the part of high-probability mutation: climbing down the steep walls of a spherical landscape (Fig. 3.2) requires little exploration, but constant exploitation - the shortest way to the optimum is that given by a hill-climbing

algorithm following the greatest height/fitness difference. Attempting to explore another point, on this type of landscape, is very likely to mean a step back, and is a (heuristically) suboptimal choice. In the case of Rastrigin's Function (Fig. 3.4), exploitation is the best strategy after exploration finds the global optimum's attraction basin.

Navigating plateaus usually means escaping them - exploitation cannot (strictly) improve the solution on a flat plateaus - or 'jumping ahead' of exploitation. This is best illustrated by Rosenbrock's Valley Function, where the optimum is at the bottom of a mostly-flat banana-shaped valley (Fig. 3.5). If the exploration tendency of the algorithm is insufficient, the GA will prematurely converge on the plateau. I believe that high-probability mutation works, in this context, by forcing the GA to take a step away from where it has converged. By flipping the vast majority of bits, a dual, Boolean-negated representation is created; if the majority of chromosomes are in close proximity on the landscape, the their dual representations are likely to be farther apart (subject to landscape properties, but true on studied function instances). By undergoing recombination and selection, the genomes are moved on the landscape; upon undergoing high-probability mutation again, they are returned to approximately similar positions - but not the same. This way, solution trajectories (for similar parts of the population) no longer need to travel along the function's surface - they can also use approximate dual landscapes to jump.

As $p_{cx}$ increases for Rosenbrock's Function (Tables 3.1 and 3.2), the performance of high-probability mutation drops (in mean); an increase in exploitation is the wrong thing to do here - the population tends to be more 'stable', with a greater inertia, and the algorithm more conservative. This slows the escape from the plateau afforded by exploration.

If the above interpretation is correct, the way high-probability mutation allows escape from local optima is through its interaction with selection. If there is a large difference in genome fitness within the population, the fitness difference in the dual population (created in the next generation, though high-probability mutation) is likely to be less. Therefore, the population will experience a greater upheaval during 'primary' generations, and a smaller change in 'dual' generations. If, however, the GA has converged to a plateau, the population will tend to remain static (with the exception of genetic drift); the 'dual' population likely shows more fitness difference between genomes, and selection in the dual generation will drive population variety for as long as the 'primary' population remains on the plateau.

Using the primal and dual selection pressures, the dual population can maintain generally poor optimum quality, serving to increase primary population variety if that variety decreases. The interplay between fitness differences (and rate of population change) in the primary and dual populations serves to automatically regulate this mechanism.

When increased exploration and population variety are not desired, the dual representation still exists and takes up computational resources, and induces a drift, however small, in the 'primary' population. This disrupts exploitation, and explains the lower performance of high-probability mutation on function landscapes where global optima have steeper attraction basins.

It is also noteworthy to mention that, due to the partial and random nature of the bit-flip induced high-probability mutation, two similar genomes can have dissimilar dual encodings. Starting from two identical representations, the probability that a random locus contains the same allele after mutation is $p_m^2 + (1 - p_m)^2$ - either it has been mutated in both genomes, or it has been unaffected in both genomes. For a $p_m = 0.95$, this probability is 0.905 - therefore, identical genomes will differ by approximately one tenth in their dual representation. This random variation can provide another path to escape premature convergence, even if the surface of the function is less favourable to the strictly Boolean-negated representation.

On the two bit-block function instances tested, high-probability mutation outperforms low-probability mutation. Considering that Genetic Algorithms are outperformed by Random Search on the GA Royal Road Function[28], the increase in exploration brought by high-probability mutation can explain the increase performance of the GA.

The same is true in the case of the GA Trap Function[13] - the trap represents a point of premature convergence, and high-probability mutation occasionally allows the GA to escape from it. It is interesting to see that high-probability mutation dual encodings do not overpower the primary encoding. The local optimum for the GA Trap Function is a bitstring of 1's, while the global optimum is a bitstring of 0's. Ideally, the Genetic Algorithm would arrive at the local optimum, then bit-flip into the global optimum; since this does not happen in the general case, primary and dual encodings seem to be in a tug-of-war, the spread of a gene being conditioned by the fitness of both its encodings.

Using high-probability mutation does not allow the GA to outperform Random Search Algorithms on the above bit-block functions - it merely improves on the baseline performance of Simple Genetic Algorithms with low-probability mutation.

### 3.2.4. Algorithmic run analysis

To better understand the behaviour of high-probability mutation and the dual encoding, a single random run of a high-probability mutation GA has been recorded and subjected to analysis. The function investigated is Rosenbrock's Function, since it has shown the best high-probability mutation comparative performance.

The run of the algorithm (which had 1000 generations) is sliced according to time, in 100-generations long slices. In order to investigate the relationship between primary and dual generations, each slice has been further divided into odd- and even- numbered generations. The maximum, minimum and average (mean) evaluation for any genome in a generation are determined, then their mean in a slice is displayed in the following table[2]:

| Gen | Odd-numbered | | | Even-numbered | | |
|---|---|---|---|---|---|---|
| no. | min | avg | max | min | avg | max |
| 1-100 | 0.29526 | 113 | 704 | 0.361683 | 115 | 731 |
| 101-200 | 0.03861 | 66 | 456 | 0.557545 | 73 | 476 |
| 201-300 | 0.08142 | 73 | 651 | 0.0754412 | 84 | 644 |
| 301-400 | 0.16268 | 68 | 564 | 0.2969455 | 76 | 630 |
| 401-500 | 0.13068 | 78 | 589 | 0.1492235 | 82 | 433 |
| 501-600 | 0.17208 | 79 | 611 | 0.126001 | 77 | 602 |
| 601-700 | 0.08712 | 71 | 461 | 0.1844335 | 71 | 424 |
| 701-800 | 0.32365 | 73 | 560 | 0.142455 | 77 | 621 |
| 801-900 | 0.07814 | 79 | 596 | 0.2248485 | 75 | 477 |
| 901-1000 | 0.1387 | 68 | 422 | 0.0972536 | 72 | 635 |
| 1-1000 | 0.13303 | 77 | 556 | 0.201871 | 80 | 575 |

Table 3.4.: GA run slicing: odd- vs. even- numbered generations.

Initially, there is little difference between odd- and even- numbered generations, and thus between primary and dual populations. The GA starts with similar fitness values for both - which makes sense, considering genomes are uniformly random-generated, and there had been little time for evolution to make significant changes.

[2]The last row displays the median for their respective column - for min, avg and max

By generation 200, a 'primary' and 'dual' difference has arisen, as shown by the minimum and average values found in odd- versus even-numbered generations. In this case, the odd-numbered generations seem to contain the primary encoding, associated with better fitness.

This relationship is reversed by generation 600, when the even-numbered generations contain better solutions; 3 other such reversals take place until the end of the algorithm.

This shows that the relationship between primary and dual representation, especially in the context of a population of genomes, suffers frequent changes and is dynamic.

While dividing representations into primary and dual is easy, since high-probability mutation offers a clear relationship between the two (partial Boolean-negation), this division is less stable during the course of a Genetic Algorithm - it is local, with respect to time. The division becomes apparent when one representation converges to a plateau or local optimum - but is less pronounced (in terms of fitness differences) otherwise.

# 4. Error Thresholds and High-probability Mutation

## 4.1. Description

In biological evolution, the Critical Mutation Rate represents the maximal mutation rate for which a gene-encoding molecule can preserve its information across generations[16]. If the mean mutation rate is kept below the critical rate, the molecule will be able to reproduce into similar descendants, creating a similarity-based cluster (or quasispecies[17], [39]), where the initial genetic information will be kept largely intact. If the mutation rate exceeds the Critical Mutation Rate, it will induce too many errors upon replication: the information contained in the group of descendants will no longer converge, on average, on the initial parent, but will start diverging rapidly.

Since, in biological evolution, mutation occurs as random errors mainly encountered in the process of information-encoding molecule (e.g DNA, RNA) multiplication, and since the transition layer between information convergence and divergence is thin, Critical Mutation Rate is also named Error Threshold: the thresholds beyond which the error rate disrupts evolution.

There is evidence that some viruses operate close to their theoretical Error Threshold[31].

Since Genetic Algorithms are inspired by biological evolution, Error Thresholds have been introduced[35] to them in an effort to augment theoretical analysis and performance predictions for GAs[33].

Error Thresholds can be used to determine optimal mutation rates[33]: by increasing mutation up to the Error Threshold (but strictly below it), the exploratory tendencies of the GA can be maximised, while still maintaining long-term evolved structures within the population.

Where an Error Threshold is placed depends on many factors, chiefly function instance and

local fitness landscapes[17], but also population size[33], generational/steady-state algorithm - in general, no feature of the GA can be entirely disregarded.

## 4.2. Determining Error Thresholds

In order to determine Error Thresholds, we need to measure the rate at which long-term structures are preserved within the population across generations. A basic way of detecting such structures is by keeping track of the majority allele on each locus, for all the genomes in the population. If the majority chances every few generations, then that locus is unstable - if most loci experience fast changes, then the population itself is unstable.

We can construct such an "average genome", as a structure similar to any genome in the GA population: it will have as many loci as any genome; each locus will have the allele that is predominant in the population at that time. This "average" or "majority" genome is called a Consensus Sequence[32]. Table 4.1 contains an example, assuming we have a very small population:

| Locus / Genome | $l_0$ | $l_1$ | $l_2$ |
|---|---|---|---|
| $g_0$ | 0 | 0 | 1 |
| $g_1$ | 1 | 1 | 1 |
| $g_2$ | 0 | 0 | 1 |
| $g_3$ | 0 | 1 | 0 |
| $g_5$ | 0 | 1 | 0 |
| consensus | 0 | 1 | 1 |

Table 4.1.: Determining Consensus Sequences - an example

In the strictest sense, if the consensus sequence of generation $t_0$ chances at generation $t_1$, then evolved long-term structures are not preserved, in their entirety, within the population. However, this instability can arise, in Genetic Algorithms, due to a rapid sprint towards a more promising area of the fitness landscape. In order to control for desirable evolution changing the population, we need to use the GA in a different way than usual.

By running the Genetic Algorithm at a constant mutation probability for many generations, we allow the population to stabilise, to reach an equilibrium state; upon changing the mutation probability (by a small amount), if Consensus Sequence changes appear, we can be reasonably

sure that the mutation probability change is responsible.

There are two ways of determining Error Thresholds with the above method - we can determine Error Thresholds from below, or from above.

Determining Error Thresholds from below involves starting at $p_m = 0$, and having initialised all the genomes with the representation of the optimum solution. By slowly increasing the mutation probability, the Consensus Sequence will eventually change. Since the population is initialised on the optimum, fitness will not drive it *en masse* away from the Consensus Sequence; the only way for it to migrate away is if the random errors induced by the high mutation rate are enough to overcome the information-correcting capacities of the GA.

The Error Threshold, then, lies between the last $p_m$ at which the Consensus Sequence was stable and the first $p_m$ where it has changed. Here, it shall be considered the highest $p_m$ for which the Consensus Sequence remains stable.

Determining Error Thresholds from above involves starting at a high mutation rate, considered much higher than the expected range for finding the Error Threshold. The mutation rate is then slowly lowered, until reaching $p_m = 0$. Analysing the succession of Consensus Sequences, we backtrack until we find the last mutation rate under which the Consensus Sequence for $p_m = 0$ remains constant. That mutation probability is considered the Error Threshold.

The advantage of determining Error Thresholds from above is that there is no need to initialise the population to the optimum - in fact, if such an initialisation were to be used, it would likely be scrambled immediately by the high error rate. In light of this fact, and since the two approaches both find the same Error Threshold values ([4] [33]), this work approaches Error Thresholds from above.

In the case of High-probability Mutation, approaching the threshold from above has to be thought in terms of effective mutation rate, not simple $p_m$. Even comparing Consensus Sequences cannot be done in successive generations, but successive even- or successive odd-numbered generation. While we approach the threshold from a higher $p_{em}$ value, in terms of $p_m$, we start at a lower mutation value (but still above 0.5), and increase it until reaching 1. Determining Error Thresholds for High-probability Mutation is otherwise similar.

## 4.3. Experimental set-up

The aim of the this experiment is to determine whether high-probability mutation preserves stable structures in the population or whether it scrambles them - fundamentally, proof by direct inspection if high-probability mutation allows for evolution to happen, or if it transforms it into a kind of Random Search. This question was raised during the anonymous review and open presentation of my paper describing high-probability mutation (N. E. Croitoru. High-probability mutation in basic genetic algorithms. *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE*, pages 301–305, September 2014).

As such, the fundamental design idea of this experiment is that of side-by-side comparison - of low- and high-probability mutation rates.

Approaching Error Thresholds (either from above or below) cannot be used to directly compare optimum-finding performance for different GA variants or parameter values. Approaching from below requires the optimum to be inserted at initialisation, while approaching the optimum from above - given the large number of generations the GA has to run at a constant mutation rate in order to stabilise - is likely to find the optimum at random.

This is why the function instances used in this experiment are a subset of the function instances investigated in Chap. 3. Taking the two together, we have information on Error Threshold values and solution quality. Not all functions have been used in this experiment, due to computational resources constraints. The functions used are:

- Rastrigin's Function, due to the comparatively poor performance(Table 3.1 of High-probability mutation.
- Rosenbrock's Valley Function, due to the comparatively good performance of High-probability mutation.
- Six-hump Camel back Function, due to the relatively similar performances of low- and high- probability mutation.
- GA Royal Road and GA Trap functions since they are bit-block functions (as opposite to the other 3 numerical functions) and Consensus Sequences provides a good visual analysis tool for the population state of bit-block functions.

Due to existing works[31],[33] on Error Thresholds calling attention to their sensitivity to population size, multiple population sizes have been investigated: 10, 25, 50, 75, 100, 200.

A crossover probability of $p_{cx} = 0.2$ has been used. Both destructive (i.e. both crossover offsprings instantly replace their parents) and non-destructive (i.e. crossover offsprings are appended to the population, existing alongside their parents) crossover has been investigated, in the attempt to evaluate its impact on Error Threshold values.

The Genetic Algorithm used has a series of modifications, allowing it to follow the procedure of approaching the Error Threshold from above:

- GAs using mirrored mutation rates are analysed in pairs, according to the relationship $p_{em} = 2 \cdot p_m \cdot (1 - p_m)$
- GAs start at $p_m = 0.2$ (for low-probability mutation) and $p_m = 0.8$ (for high-probability mutation) - at $p_{em} = 0.32$, where a third of the bits in the population are randomly scrambled every 2 generations. The GAs run at these mutation rates for 20000 generations, in order to reach an equilibrium state.
- $p_{em}$ is lowered in both mirrored runs; low $p_m$ decreases in 0.001 steps until $p_m = 0.01$ is reached. Afterwards, it is decreased by 0.0001. High $p_m$ increases in 0.001 steps until $p_m = 0.95$ is reached. Afterwards, it increases in 0.0001 steps. The GA is allowed to run 2000 generations at each mutation probability.
- The GA stops upon $p_m$ reaching either 0 or 1.
- The Error Threshold for low-probability mutation is found by starting at $p_m = 0$, and moving upwards along the Consensus Sequences until the first Consensus Sequence change.
- The Error Threshold for high-probability mutation is found by starting at $p_m = 1$, and moving downwards, until the first Consensus Sequence change.
- A 5-genome elitism is used: the best 5 genomes from each generation are guaranteed to be carried over to the next. This change was implemented because non-elitist GAs used could not show stable Consensus Sequences: very small changes in the otherwise-stable Consensus Sequences would occur, no matter the mutation rate.

## 4.4. Results

Lists of Consensus Sequences can be easily represented visually[32]. A Consensus Sequence is a string of bits; a 0 can be represented by a black pixel or square, a 1 by a white pixel or square. Having multiple Consensus Sequences one after another (arranged perpendicularly to the direction of the bitstring), ordered by time or mutation probability, offers a good way to visualise the evolution of the population.

Since many population sizes investigated are even numbers, there exists the possibility of ties - when, for a certain locus, the number of 0's in the population is equal to the number of 1's. In order to simplify visualisation, there is a rule in place to decide the color of such loci:

- At the start of the algorithm, all positions in the Consensus Sequence are 0 (black).
- In the event of a tie, the Consensus Sequence color for a certain locus remains unchanged.

Simply, if a majority is not reached, we preserve the majority decision from previous generations.

The example figure below (Fig. 4.1) serves to illustrate the main elements of a double Consensus Plot:

- the horizontal divide in the middle of the plot separates the two subplots - above, the search for Error Thresholds for high-probability mutation; below, for low-probability mutation.
- the vertical axis represents the mutation probabilities of the GA. These mutation probabilities also index the Consensus Sequences in a temporal order: the GA run starts at $p_m = 0.2$, then progresses to $p_m = 0$, for the lower half of the plot. For the upper half of the plot, Consensus Sequences at $p_m = 0.8$ are the oldest, and those at $p_m = 1$ are the newest.
- the the horizontal axis $l$ represents positions on the solution representation, loci on the genome. When representing fixed-point numbers (as opposite to bit-block function candidate solutions), the least-significant bit of the first number is the leftmost. The number of bits in a genome are displayed at the far right of the axis.

The name of the function instance and the population size used are shown in the caption. Numerical functions have been used in their 2-dimensional, $10^{-5}$ precision instances. Bit-block

functions have been used in $8 \times 8$ instances - 8 blocks of 8 bits each.

On visual examination of the Consensus Plot (Fig. 4.1), we can see the Error Threshold for low-probability mutation (at $\approx 0.005$) and for high-probability mutation (at $\approx 0.97$).

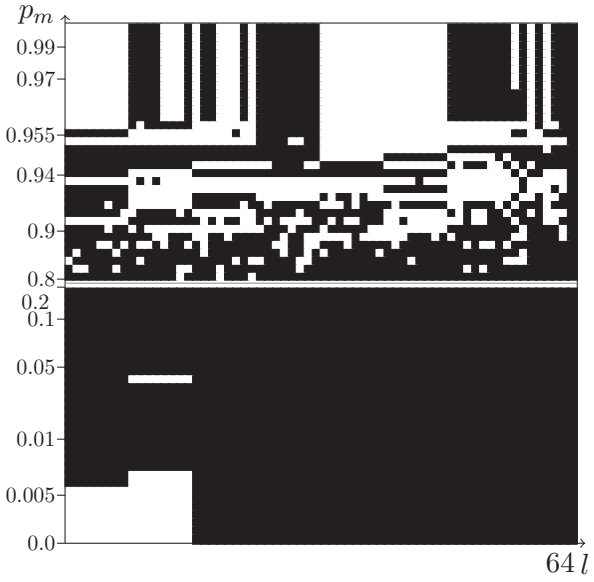


Figure 4.1.: Example Consensus Plot: Trap Function $8 \times 8$, pop 75

In the interest of space and clarity, only Consensus Plots for population sizes of 100, for each function instance, are included here. The full set of Consensus Plots, encompassing all the data produced by this experiment, can be found in Appendix A.

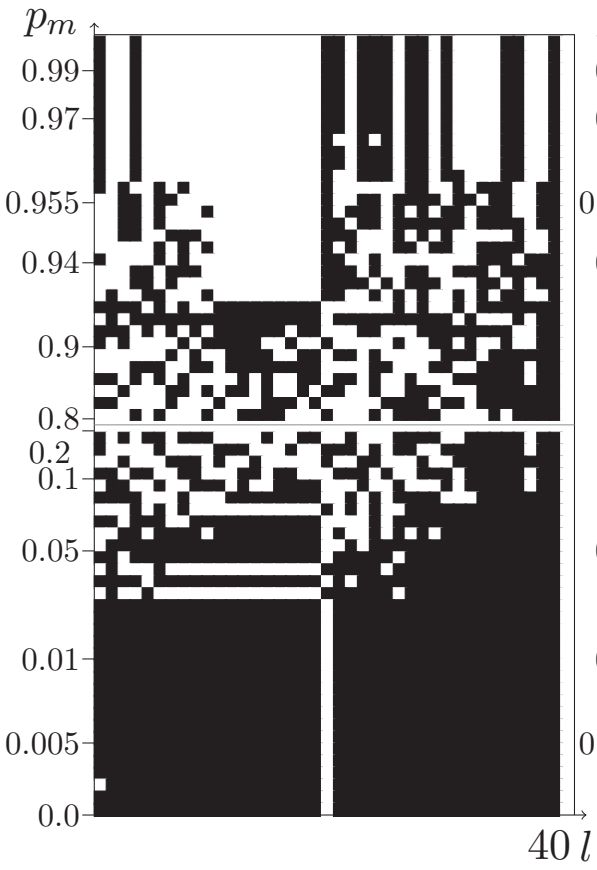


Figure 4.2.: Consensus Plot: Rastrigin's Function, pop 100, destructive cx

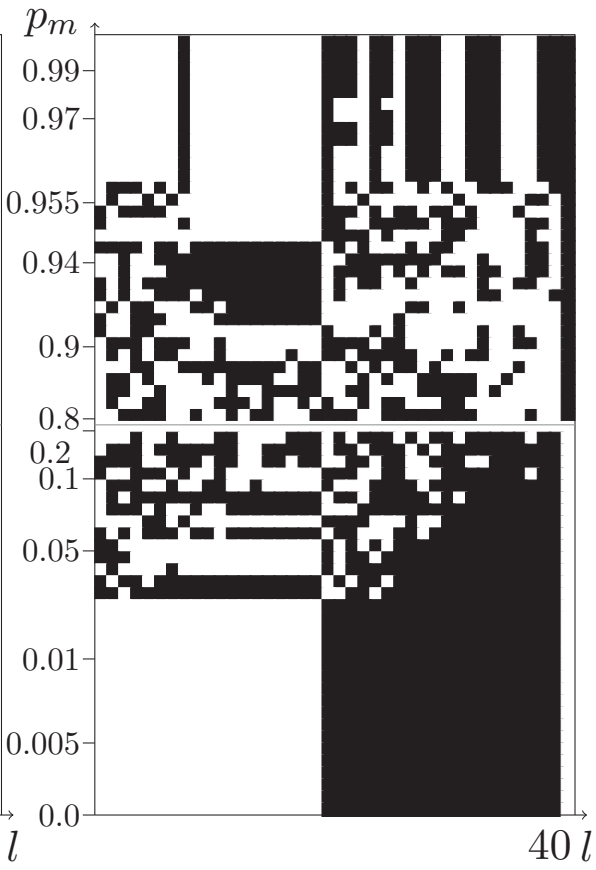


Figure 4.3.: Consensus Plot: Rastrigin's Function, pop 100, nondestructive cx

It is interesting to note that the optimum can be found at $x_1 = 0, x_2 = 0$, which, in the consensus sequence, has the closest representations of $0000\ldots1$ or $111\ldots0$ - either a black block followed by a white thin line, or a white block followed by a black thin line.

In the case of Rastrigin's Function, both low- and high-probability mutations have similar Error Thresholds, mirror-wise. The exception is low-probability mutation in Fig. 4.2, where a single, least significant bit flip lowers the Error Threshold.

The majority of the population converges to optimal representations in the case of low-probability mutation. In contrast, the population under high-probability mutation does not converge to a Consensus Sequence that is a representation of the optimum solution.

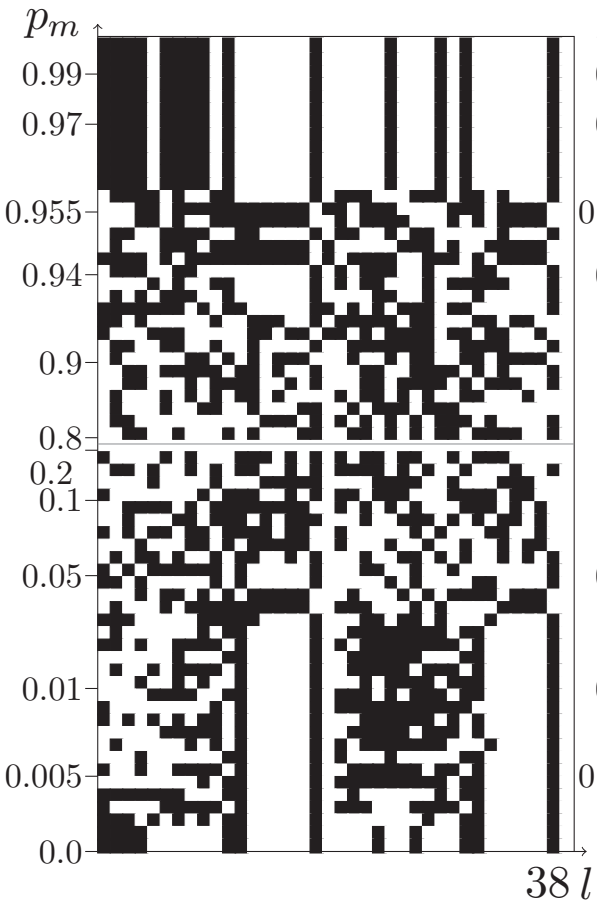


Figure 4.4.: Consensus Plot: Rosenbrock's Function, pop 100, destructive cx

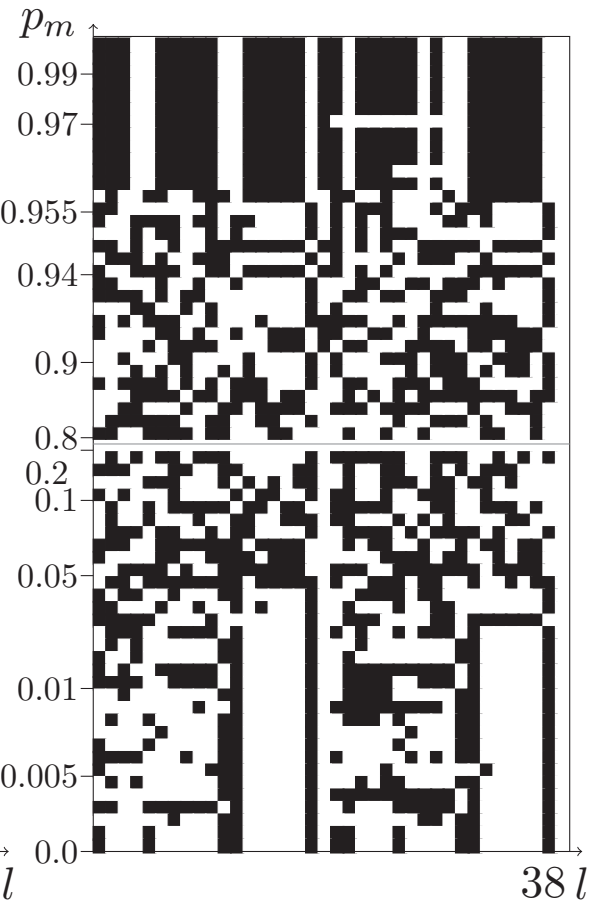


Figure 4.5.: Consensus Plot: Rosenbrock's Function, pop 100, nondestructive cx

In the case of Rosenbrock's Valley Function, high-probability mutation has a better Error Threshold than low-probability mutation (comparing them through equivalent $p_{em}$ values). For low-probability mutation, it is interesting to see that the most significant bits are the first to get locked in place and the Consensus Sequence fluctuations then happen for the less significant bits.

Nevertheless, the low Error Threshold value for low-probability mutation is not, in this case, due to single-bit flips; large parts of the Consensus Sequence are unstable.

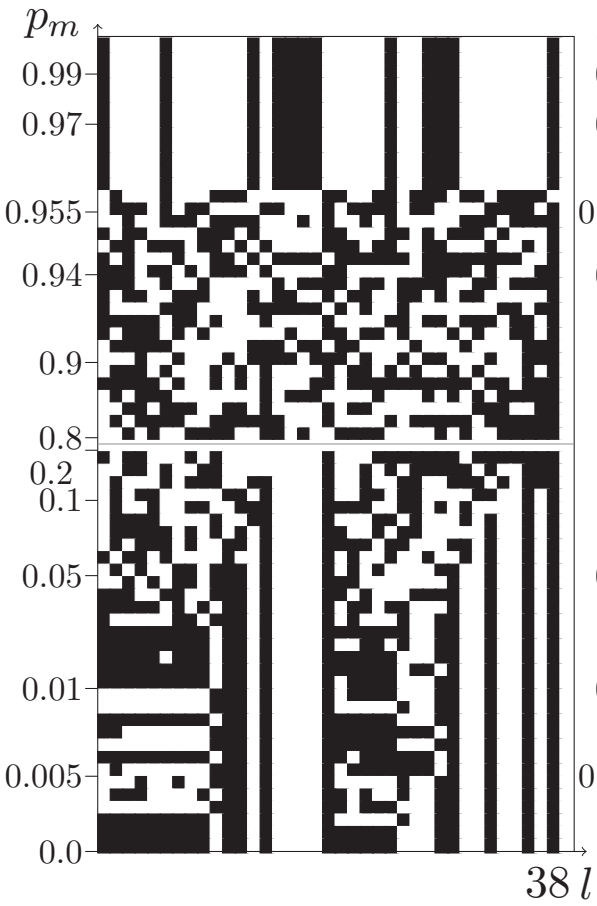


Figure 4.6.: Consensus Plot: Six-Hump Camelback Function, pop 100, destructive cx

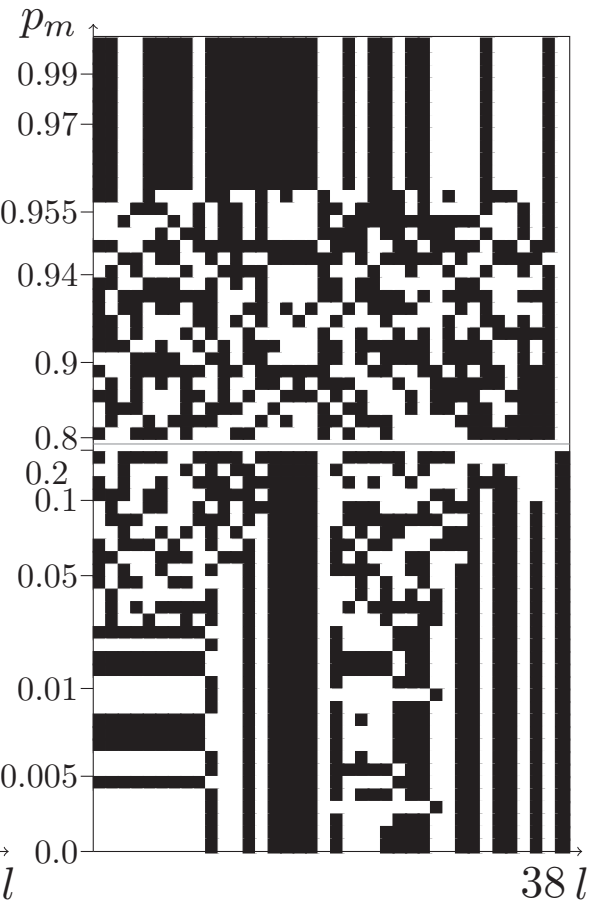


Figure 4.7.: Consensus Plot: Six-Hump Camelback Function, pop 100, nondestructive cx

In the case of low-probability mutation for the Six-Hump Camelback Function, the Error Thresholds are low. The Algorithm first fixes significant bits in place, then proceeds to less significant bits.

The phenomenon is also visible in high-probability mutation (Fig. 4.7), but to a smaller degree. Rather, in this case, the switch to the final Consensus Sequence tends to happen simultaneously.

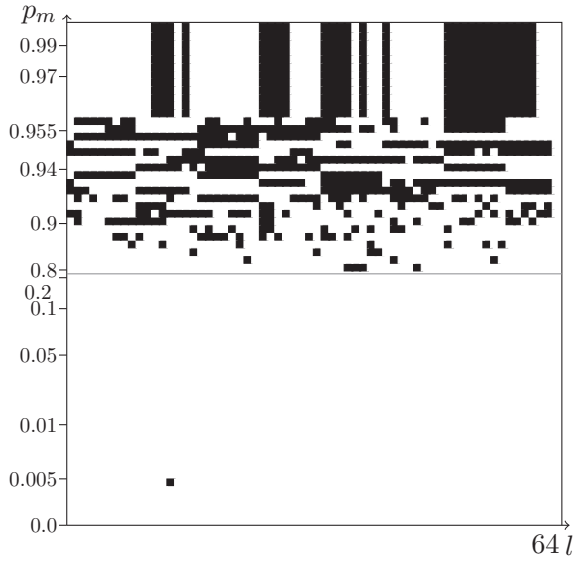


Figure 4.8.: Consensus Plot: GA Royal Road Function, pop 100, destructive cx

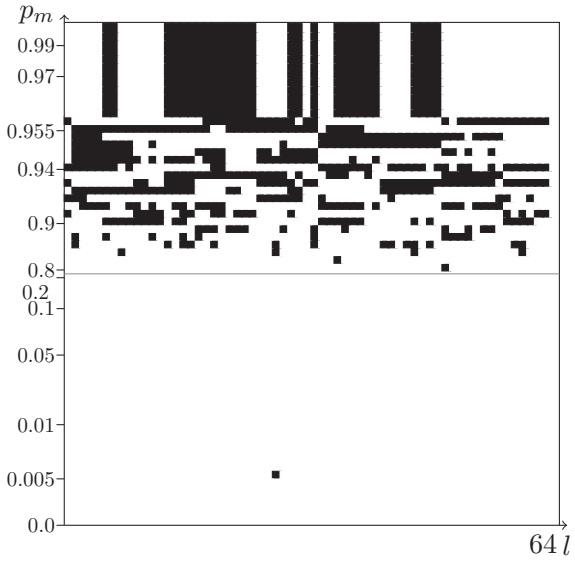


Figure 4.9.: Consensus Plot: GA Royal Road Function, pop 100, nondestructive cx

On the Royal Road Function, low probability mutation Consensus Sequences quickly converge towards the optimal genome. High-probability mutation converges to a different Consensus Sequence.

There are (Fig. 4.8, 4.9) low-$p_m$ bit flip in the consensus sequence which lower the Error Thresholds for low-probability mutation.

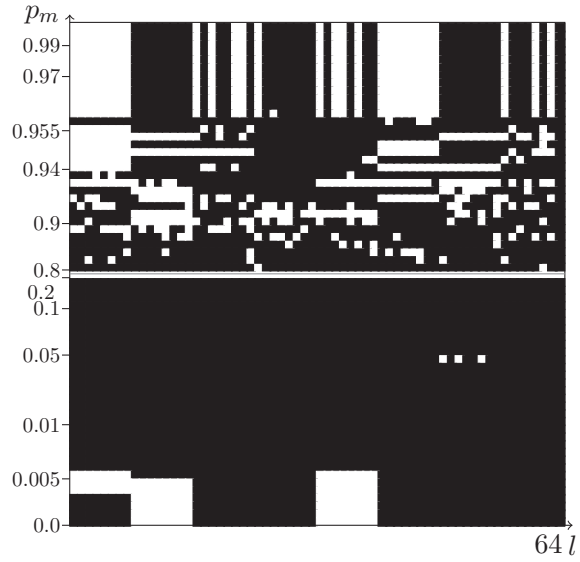


Figure 4.10.: Consensus Plot: GA Trap Function, pop 100, destructive cx

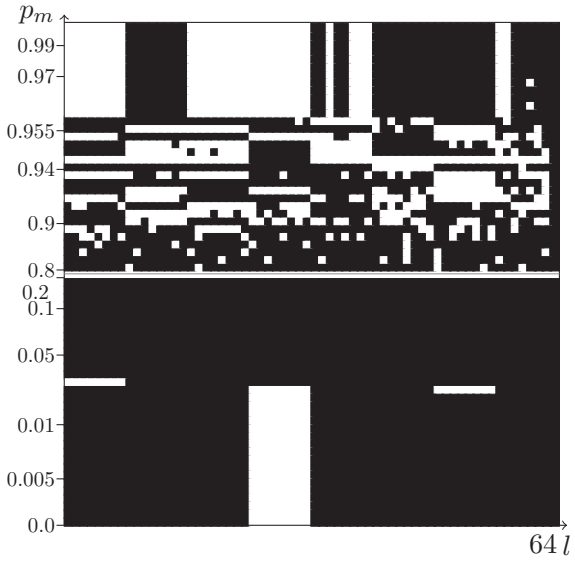


Figure 4.11.: Consensus Plot: GA Trap Function, pop 100, nondestructive cx

For the GA Trap Function, the low-probability mutation GA converges mostly into the trap (a genome of $0 \ldots 0$), and only manages to escape it for a few blocks.

By contrast, high-probability mutation escapes the trap more frequently.

The following plots bring together the above results, together with those in Appendix A. They show the mutation probabilities where the Error Threshold occurs for each function instance, mutation type, population size and cross-over variant used.

Solid lines and squares correspond to non-destructive cross-over, while dashed lines and triangles to destructive cross-over. The lines have been linearly interpolated, while the vertices of the plot represent experimental data.

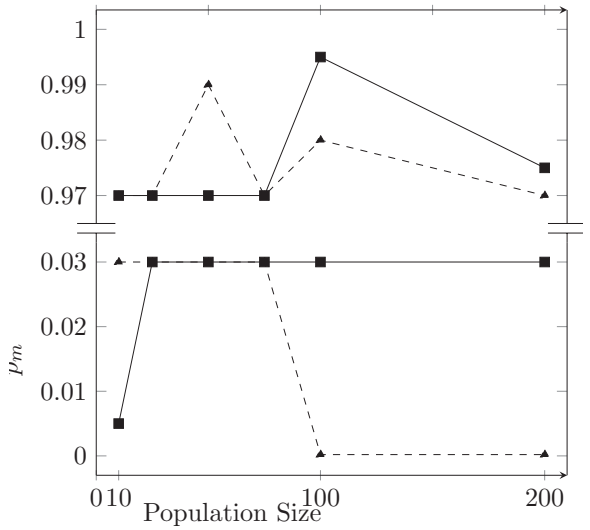


Figure 4.12.: Error Thresholds: Rastrigin's Function

In the case of Rastrigin's Function(Fig. A.2-A.11), both types of mutations have comparable performances. It is interesting to see that increasing population size tended to worsen the threshold in all cases but low-probability mutation, non-destructive cross-over.

Destructive cross-over lowers the Error Threshold in some situations, although this is not a general tendency.

Except for the above-mentioned disturbing factors, Error Thresholds for both mutations tended to be constant and set at the same $p_{em}$ value ($p_m = 0.03$ and $p_m = 0.97$).

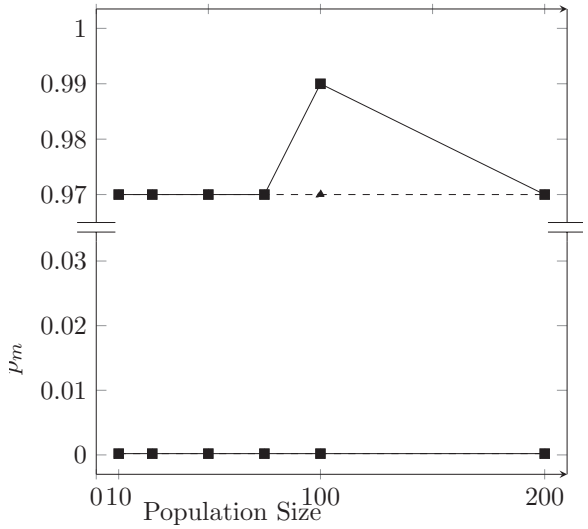


Figure 4.13.: Error Thresholds: Rosenbrock's Function

In the case of Rosenbrock's Valley Function (Fig. A.14-A.23), the Error Thresholds for high-probability mutation are much better than for low-probability mutation (where it was consistent at 0.002).

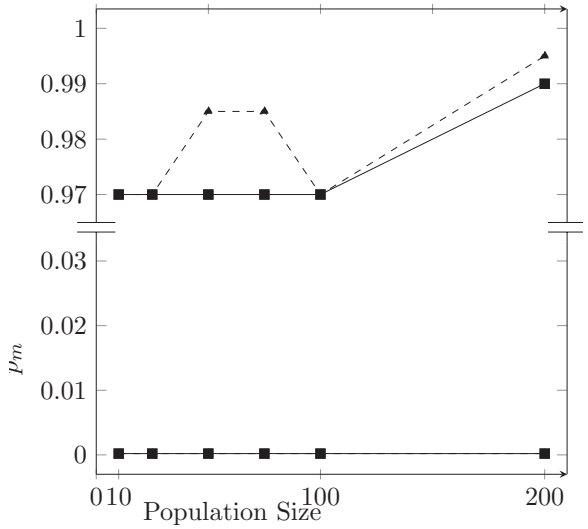


Figure 4.14.: Error Thresholds: SixHump Camelback Function

For the SixHump Camelback Function(Fig. A.26-A.35), high-probability mutation has bet-

ter Error Thresholds. At the same time, a large population size adversely impacts the Error Threshold.

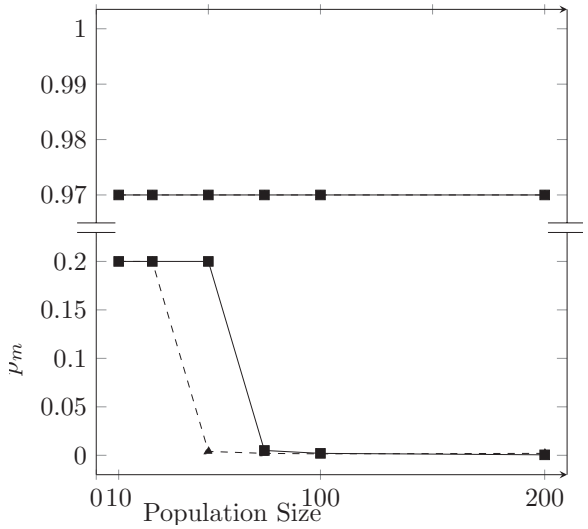


Figure 4.15.: Error Thresholds: RoyalRoad Function

Despite generally converging to the optimal representation, the Consensus Sequences for the GA Royal Road (Fig. A.38-A.47), low-probability mutation, exhibits small variations which lower Error Thresholds as population size increases.

Nevertheless, Error Thresholds for low-probability mutation and small population sizes are extremely high (at mutation probabilities of at least 20%, the starting point of the approach towards the Error Threshold) - another argument showing that Random Search Algorithms tend to perform better than Genetic Algorithms on this function.

High-probability mutation experiences constant error thresholds, no matter the population size or cross-over type.

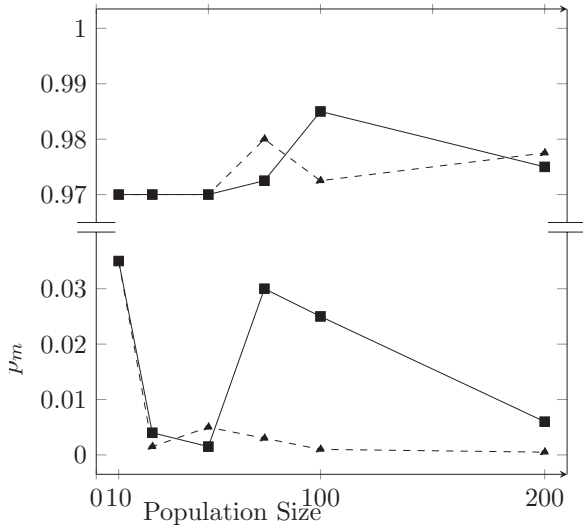


Figure 4.16.: Error Thresholds: GA Trap Function

The results for the GA Trap Function(Fig. A.50-A.59) are the most erratic; high-probability mutation, having better results (Table 3.3) is more consistent, while low-probability mutation experiences large Error Threshold variation.

## 4.5. Conclusions

By allowing stable Consensus Sequences to exist, high-probability mutation differentiates itself from simple random search happening within a Genetic Algorithm. Even if the majority of bits in the population experience binary negation every generation, the Consensus Sequence returns to its previous form every other generation. The large-scales structures that are formed by the process of evolution persist and progress under high-probability mutation.

By visually observing Consensus Plots, a few differences can be noticed between the two mutation variants. In some cases (Fig. 4.4, 4.5, 4.6, 4.7)), low-probability mutation fixes the more significant bits of the Consensus Sequence ahead of time, while the less significant bits are kept variable for longer. In contrast, high-probability mutation tends to freeze all the bits in the Consensus Sequence at once.

I interpret this as low-probability mutation locking into a smaller area in the function landscape, and, through exploitation, attempting to find the optimum. High-probability mutation,

through increased exploration, "stumbles upon" a sufficiently favourable family of similar solutions (or a quasispecies), onto which it locks. It is worthwhile to mention again that, due to the very large number of generations spent approaching the Error Threshold from above, the optimum is very likely to be found early: the Consensus Plots are a good way to see the tendencies of the algorithm in an equilibrium state, not to measure the optimum-finding speed of the GA.

For the Royal Road Function (Fig. 4.8, 4.9), the low-probability mutation Consensus Sequences converge quickly towards representing the optimum (a bitstring of $111\ldots1$ - entirely white in the Consensus Plot). In fact, the Consensus Sequences are stable at extremely large effective mutation values ($p_m \approx 20\%$) - values much too large to allow stable mutation to take place in the other functions examined. Consensus Sequence change away from the optimum occurs, in fact, at low mutation rates ($p_m \approx 0.5\%$). I interpret the above observations as proof for Random Search Algorithm's better performance compared to Genetic Algorithms, on the GA Royal Road Function. Mutation rates of $\approx 20\%$ bring in too much entropy to each genome, and, in general, scramble most information evolved within the population. This randomisation, coupled with the fitness function still guiding the search towards the optimum, allows the Genetic Algorithm to approach the behaviour and traits of a Random Search Algorithm. When applied on function instances where RSAs have low performance, a high-entropy mutation lowers the GA performance; however, when applied on problems where RSAs have good performance, the hybridised GA's performance increases.

In contrast, high-probability mutation, being a low-entropy, low effective-mutation (Fig. 3.1) operator, does not scramble the evolved information contained in the population. What the Consensus Plots (Fig. 4.8, 4.9) show is that the Error Thresholds occur relatively early ($p_m \approx 0.96$) but the stable Consensus Sequence is not a representation of the optimal solution. Coupled with the high-probability mutation's better performance (compared to low probability mutation, Table 3.3), this leads me to conclude that high-probability mutation maintains a heterogeneous population, exploring multiple regions of the landscape.

It is this exploratory behaviour that allows high-probability mutation to escape some of the traps in the GA Trap Function (Fig. 4.10, 4.11), while the low-probability mutation converges, in most cases, away from the optimum ($111\ldots1$) and into the trap ($000\ldots0$). I find the trap-escapes of the low-probability mutation to be random and difficult to predict, as seen from the large range of values for the Error Threshold in this case (Fig. A.49-A.60).

Overall, Error Thresholds are sensitive to a many factors:

- Population size increase, in general, lowers the quality of Error Thresholds. This could be explained, in part, as diluting the impact of the 5-genome elitism introduced in the GA.
- Destructive crossover, in general, lowers the quality of found thresholds. Since destructive cross-over implies that recombined genomes immediately replace their "parents", it makes it easier for the population to migrate away from the dominant quasispecies - from the majority of individuals having largely similar genes. By contrast, non-destructive cross-over does not delete the "parents", allowing the dominant quasispecies a good chance at replication, in the context of favourable fitness values.
- Mutation type: Error threshold behaviour is, in all cases, different between high- and low-probability mutation. The two mutation types affect the whole algorithm, making it behave in different ways - but they are both forms of evolution, since Error Thresholds are identifiable for both.
- Small fluctuations: especially in the case of low-probability mutation, Error Thresholds are worsened by small (1-bit, 1-mutation epoch) fluctuations which quickly return to the previously stable Consensus Sequences (e.g. Fig. 4.2, 4.8, 4.9). Although, in the strictest definition, these fluctuations do lower the Error Threshold, relaxing the constraints for identifying Error Thresholds (to ignore small, temporary fluctuations) would allow us to better identify the phase transition between evolution and random search. The small fluctuations do not represent an erasure of the accumulated, evolved information within the population.

# 5. Lowering overfit in Neuroevolution

## 5.1. Description

There are many problems approachable by way of Genetic Algorithms; by encoding bitstrings, the GA can search patterns; by encoding real-valued parameters (as fixed-point binary numbers), numerical optimisation tasks can be undertaken.

By encoding the synaptic weights of Artificial Neural Networks, Genetic Algorithms can evolve ANNs for specific tasks, in a flexible and adaptive way.

Artificial Neural Networks - specifically multilayer ANNs - are know to be universal approximators[21]; however, training ANNs using Backward Propagation of Errors, alongside a Gradient Descent Algorithm is prone to numerous limitation.

By using a Genetic Algorithm to evolve the weights of an ANN, some degree of flexibility is gained: precise sets of input-output pairs are replaced by a fitness function, there is no need for the neuron activation function to be differentiable[1]. GAs can more easily escape from local optima and plateaus than Gradient Descent algorithms, and are less likely to learn the order of inputs, instead of the information contained in inputs.

However, ANNs and GAs are prone to overfit - and so is the combined method. The application of high-probability mutation is aimed at reducing overfit, through its increase in exploration and ability to escape local optima and plateaus. In order to measure the impact of high-probability mutation on overfit, it is compared, side-by-side, in terms of optimum quality, with a similar algorithm using low-probability mutation.

The problem solved using Genetic Algorithms and Artificial Networks - Neuroevolution - is predicting the evolution of a real-world Internet social network.

[1] In BackPropagation[15], the partial derivative of the error (between network output and expected output) is computed, relative to the weights of the network. This requires the activation function to be differentiable.

### 5.1.1. Motivation

I approached this problem due to multiple reasons. I felt that, beyond traditional benchmark problems, High-probability Mutation could use testing in a real-world problem.

This particular instance was familiar to me, having studied it in my M.Sc. thesis[6]: Neuroevolution seemed like a good way to approach the problem - in addition to the advantages of Genetic Algorithms over Gradient Descent Algorithms, ANNs - and, especially, multi-layer[5] ANNs - are know for their universal approximator[21] and predictive[42] qualities.

One of the issues I found was the relatively high rate of overfit. Given the properties of high-probability mutation discussed in Chapters 2 and 3, it seemed likely it could lower that overfit.

### 5.1.2. Problem description

The problem consists in predicting the future state of a social network, from records of its past states. The studied network is Flickr[2], a social network oriented towards uploading, viewing and sharing photographs.

Through its public API, a series of large-scale statistical measures about the network can be obtained. In this case, the data consists of 10 successive snapshots of the most popular tags on the network, taken at fixed time intervals.

The problem goal is to predict the 10-th snapshot from the previous 9.

More specifically, the terms used here mean:

- Tag: an image can be described using multiple textual terms or short phrases, assigned to it by humans or by automated metadata tagging. Automated metadata tends to be geographic location and camera technical specifications. Human-assigned metadata is aimed at describing the contents of the image; users can assign tags when uploading or viewing images, and are mostly free in entering text strings as tags - thus, the range of possible tags is quite large.
- Popular Tag: at certain times, some tags are assigned more than others. For instance, in the event of an eclipse, there are likely many pictures of the eclipse uploaded, and the tag “eclipse” is likely to be popular - at least in certain parts of the world. The most popular tag at a certain moment receives a score of 1, while other tags, having received less

assignments in the recent past, receive proportionally lower scores. For example, if the tag "eclipse" receives a score of 1 for 3000 assignments, the tag "sun", with 1500 assignments receives a score of 0.5. Another tag, with 2983 assignments would receive a score of 1, as well. The score is accurate to the percent (the API delivers integers between 100 and 0, which are scaled within the $[0, 1]$ range).

- Snapshot: the list of the 200 most popular tags at a certain time, along with their scores. Two successive snapshots do not contain all the same tags - since the network evolves in time, untracked tags could become popular enough, and popular tags could become obscure. In practice, for this data set, the number of tags exceeds 340.
- Data set: the time-ordered list of all 10 snapshots. Each successive snapshot is taken 12 hours after the last.

Since many tags don't appear in all snapshots, a pre-processing step was necessary: if a tag has an unknown score at a certain time, it is assigned a score lower-or-equal to that of the least popular tag. If $ls$ is the lowest score in a snapshot, a tag with an unknown score would receive one of $max(0, ls - 0.01)$.

Solving the problem involves lowering the prediction error - that error is measured, specifically, in terms of Mean Absolute Error:

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |{}_p tag_i - {}_r tag_i|$$

where $n$ is the number of tags in the whole data set, ${}_p tag_i$ is the predicted value for the $i^{th}$ tag, and ${}_r tag_i$ is the recorded value of the $i^{th}$ tag.

The MAE was chosen over other error computing measurements because it can be translated directly into tag assignments mistakenly predicted by the algorithm.

Since tag scores are assigned, in a single snapshot, in a linear way with respect to actual tag assignments, and since the algorithm predicts a single (the last) snapshot, a linear error function offers a number that translates linearly into tag assignments. We need only multiply the MAE with the total number of tag assignments at a certain time, to obtain the total number of "clicks" (human actions) the algorithm mis-predicted. Even if the total number of tag assignments is unknown (as the public API does not provide that), we can replace it with an arbitrary constant,

and be certain the MAE is a linear factor of it.

## 5.2. Method

The Neuroevolution algorithm is trained on the first 9 snapshots. The ANN evaluation consists in feeding the network the first snapshot, from which it predicts the second. The MAE between the prediction and recorded second snapshot is computed, then the ANN predicts the third snapshot from the second, and so on.

The MAE for the final, 10-th snapshot prediction is computed. However, it is not part of the evaluation for the ANN and the genome during the Genetic Algorithm run - it doesn't influence fitness. Instead, it is externally reported by the Genetic Algorithm framework. This is an essential step for proving overfit happens, and for measuring it, as is detailed below.

For clarity, the evaluation function structure for a genome is written below, in pseudocode:

```
function evaluate(genome):
  ANN = decode(genome)
  totalError = 0
  for i in 1..8:
    past = snapshots[i]
    future = snapshots[i + 1]
    prediction = ANN(past)
    error = MAE(future, prediction) * i^2
    totalError += error
  return(totalError)
```

Predictive errors for older snapshots are given relatively less importance in computing the fitness of the ANN. This is done by multiplying the MAE with the square of the snapshot index (which goes from 1 to 8, increasing as a function of the real-world time when a snapshot was taken). $index^1$, $index^2$ and $index^3$ were compared, in a small-scale test, and $index^2$ was found to give the best results, in terms of final prediction.

The goal is the prediction of the 10-th snapshot from the 9-th; due to temporal locality, it is likely that the most similar predictive model learnt from a single snapshot pair is produced

from the transition from snapshot 8 to 9 - and errors appearing in predicting the 9-th snapshot should be given the greatest importance. However, past transitions need to be taken into account - ignoring them will likely overfit the ANN to the training data, and lower its predictive capacities. There are phenomena not appearing during the transition from snapshot 8 to 9, but present in older information, which still could make an appearance during the transition from 9 to 10. As a general rule, temporal locality merely lowers the probability of older phenomena appearing - while being given less importance than recent phenomena, they still need to be taken into account.

The particular properties of the algorithm are detailed below:

Artificial Neural Network:

- Structure: the ANN is a complete feedforward network, with 2 hidden layers. Two hidden layers are used because it is the smallest kind of network known to be an universal approximator[21] and offers good performance[5].
- Size: The input and output layers are the same size, the number of unique tags in the first 9 snapshots. The size of each hidden layers is the same. While it is known that ANNs are universal approximators - i.e. they can approximate any function arbitrary well - the minimum network size and complexity to approximate a function is not known, for the general case. This is why determining ANN hidden layer size was treated as exploring a right-unbounded interval: starting with a hidden layer with $2^1$ neurons, the size is doubled with each new exploratory step, until good performance is judged to be found. In this case, the exploration goes up to $2^8$ neurons on each hidden layer.
- Neurons: each node of the Artificial Neural Network is composed of a perceptron; its activation function is the logistic sigmoid $f(x) = \frac{1}{1+e^{-x}}$. Despite the possibility for Neuroevolution algorithms to use non-differentiable activation functions, this choice was found to function well enough, and is supported by theoretical results[10], [11].
- Weights: the weights of the ANN are fixed-point real-valued numbers, with a guaranteed precision of $10^{-5}$.

Genetic Algorithm:

- Fixed-length binary-encoded genomes: a genome encodes a single ANN, and it stores the weights of that network.
- Genome size: the number of weights depends on the number of tags and the size of the hidden layers. Assuming 300 tags and $2^5$ perceptrons in each hidden layer, a genome encodes $300 \cdot 2^5 + 2^5 \cdot 2^5 + 2^5 \cdot 300 = 20224$ weights.
- Crossover: single point, position-independent, non-destructive (i.e. crossover 'offsprings' do not immediately replace their 'parents', but join them in the population). $p_{cx} = 0.3$.
- Population size: a constant size of 100 genomes was used. Selection serves to lower population size after non-destructive crossover.

Since the goal of this experiment is too see how high-probability mutation affects the algorithm, compared to low-probability mutation, the experiments are mirrored, with the only non-random variable changed being the mutation probability. Two such probabilities are compared, side-by-side: 0.01 and 0.95.

## 5.3. Determining overfit

In order to assess the impact of high-probability mutation on overfit, we need a way - however partial or approximate - to measure or find overfit.

Poor predictive performance for an Artificial Neural Network can have multiple reasons; we can estimate these causes by comparing ANN performance on training set data vs. test data.

If the ANN has poor performance on the training set, and poor performance on the test data, then it does not model the problem well enough; perhaps the complexity of the problem is more than can be encompassed by an ANN of that size and structure, or the method we're using is flawed.

If the ANN has good performance on the training set, and good performance on the test data, then our approach is good, and training allows the ANN the ability to generalise.

If the ANN has good performance on the training set, but poor performance on the test data, then the ANN has been fit to the training data overly much, limiting its ability to generalise and adapt to new data.

Thus, a way of determining overfit is comparing the performance fall-off of the ANN, from training to test. There are some issues with this approach, one of them being the fact that we compare the same model for slightly different data. Our measure for overfit delivers an accurate answer only if the data is fundamentally similar. This issue can be mitigated by randomly choosing the training and test data. Since we have limited and time-sorted data, this method cannot be easily applied here.

The fact that the GA reports both the traditionally-proposed candidate solution (henceforth named proposed solution) and the best solution ever evaluated during that algorithm run (henceforth referred to as best solution) allows for another possible measure for overfit:

The best solution represents proof by inspection that a solution at least as good is possible - it sets an empirical lower bound on solution quality. Specifically, it proves that an ANN with that structure and size can predict the future snapshot with, at least, the same MAE as the best solution.

The proposed solution cannot be better than the best solution in the same GA - at most, the proposed and best solutions are identical in MAE.

By comparing the difference in prediction quality between the best and proposed solution, we can encounter two situations. If the proposed solution has the same MAE as the best solution, then we have no direct proof of overfit occurring. If the proposed solution has worse MAE, then the difference in predictive errors is attributed to overfitting: the GA chose, from among all solutions it had visited, a suboptimal one. Since the GA makes decisions based on fitness values derived from the training data, the mistakes it makes in choosing ANNs that are good for the training step but bad for the prediction step are overfit mistakes - it over-adapts on the training data.

This difference in prediction quality between the best and proposed solutions serves as the empirical measure of overfit. It is a direct comparison of ANN performance on the same data, as opposite to slightly different data (i.e. the training and test sets), and is ideal for the current data set, where there are few data snapshots, and we cannot rely on randomisation to smooth out this difference.

While this method proves (by empirically finding it) a certain lower bound for overfit, it accounts for merely a part of overfit - it sets no upper bound. When reporting overfit reductions, this work is limited to inspecting that empirically-measured part of the overfit.

## 5.4. Results

Due to limits on computational resources, this experiment only investigates ANN hidden-layer sizes up to $2^8$ neurons. Furthermore, while Neuroevolution runs up to hidden-layer sizes of $2^7$ have associated sample sizes of 32, for $2^8$, the sample size is just 4. Only the vertices of the plots have been sampled, and each represents a median; the lines connecting them have are linearly interpolated, to connect sets of similar results.

The shorthand terms *proposed* and *best* have the following meanings:

- Proposed: the solution proposed by the Genetic Algorithm, as the best-evaluated (on the training set) in the last generation.
- Best: the best solution visited by the Genetic Algorithm in a run, as the best-evaluated on the test data in all generations.

| Hidden layers size ($log_2$) | Mutation rate = 0.01 | | Mutation rate = 0.95 | |
|---|---|---|---|---|
| | Proposed | Best | Proposed | Best |
| 1 | 0.445267 | 0.431208 | 0.445989 | 0.431974 |
| 2 | 0.443335 | 0.431202 | 0.443494 | 0.431860 |
| 3 | 0.447829 | 0.432035 | 0.441595 | 0.431310 |
| 4 | 0.440724 | 0.429161 | 0.439934 | 0.429842 |
| 5 | 0.428891 | 0.414200 | 0.427181 | 0.414853 |
| 6 | 0.416323 | 0.370370 | 0.409083 | 0.379290 |
| 7 | 0.348288 | 0.307266 | 0.338792 | 0.317874 |
| 8 | 0.249078 | 0.136471 | 0.230428 | 0.153046 |

Table 5.1.: Neuroevolution results

The experimental results do show an improvement in proposed solutions found as high-probability mutation is used (compared to low-probability mutation).

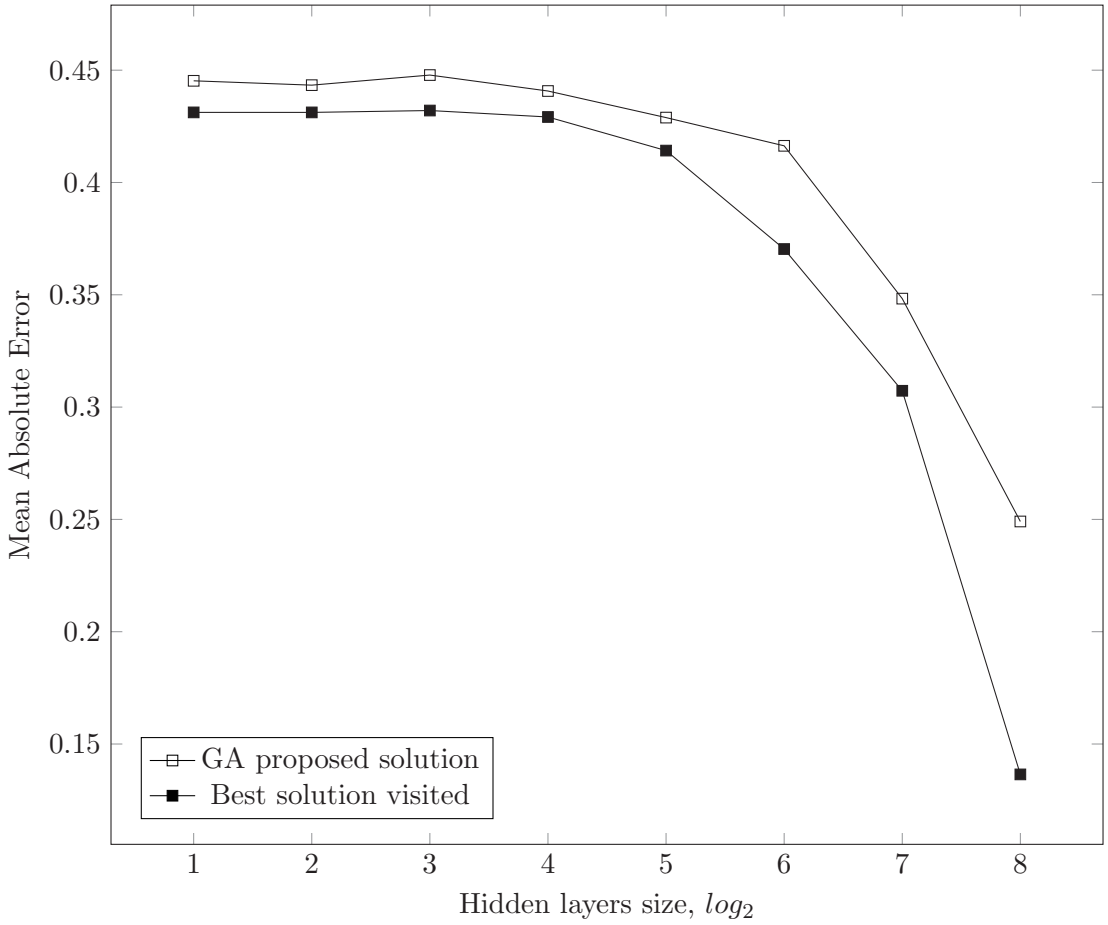


Figure 5.1.: Neuroevolution $p_m = 0.01$ results

MAE decreases as ANN size increases: the data is quite complex, and it requires large ANNs, capable of approximating complex relationships. Better results are found only with ANN hidden layer sizes of above $2^6$. Results are largely unchanged between ANN sizes lower than that.

Overfitting is an issue: the best predictive models visited by the GA have MAE values below 0.15, and overfit increases, comparatively, with ANN size.

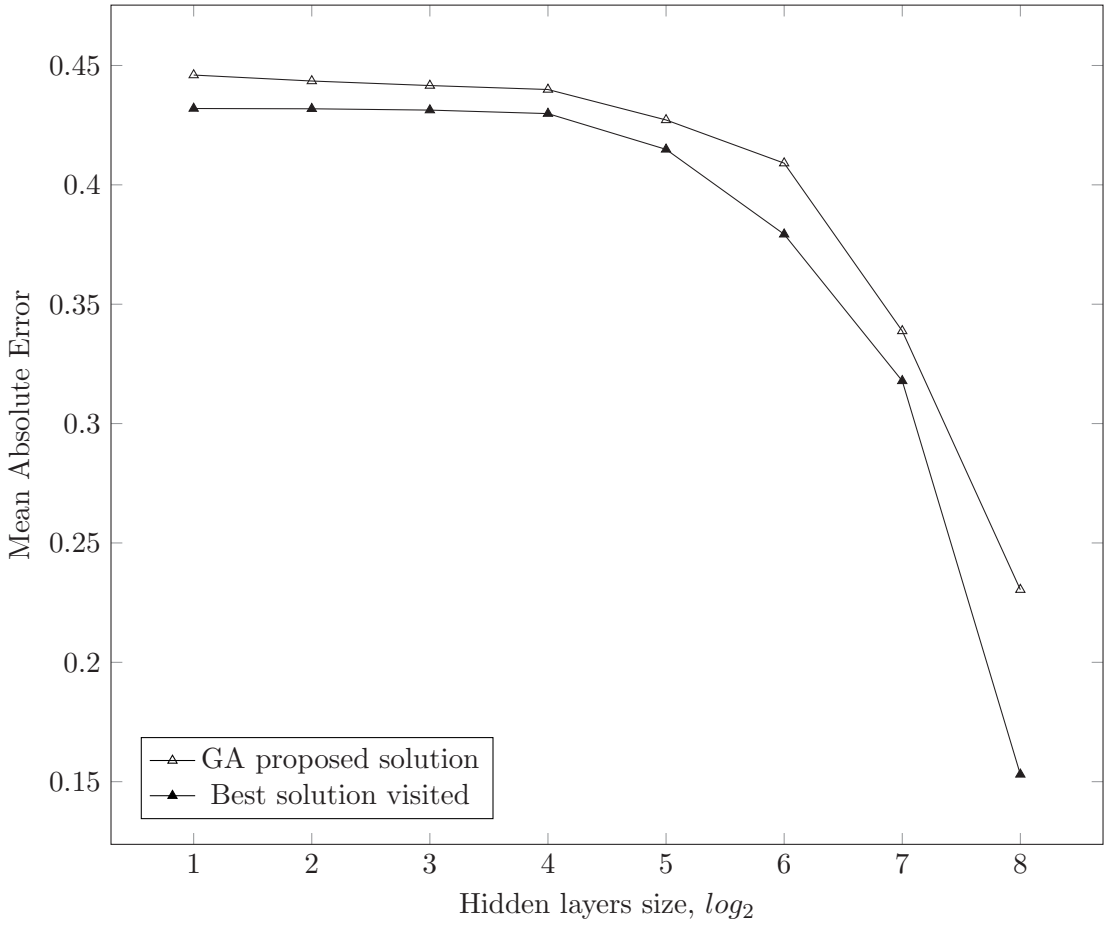


Figure 5.2.: Neuroevolution $p_m = 0.95$ results

A GA using the mutation probability of 0.95 shows the same general behaviour: error improves as ANN size increases, but overfit increases. However, the distance between the Best solution visited and the GA proposed solution is reduced through the use of high-probability mutation, compared to low-probability mutation.

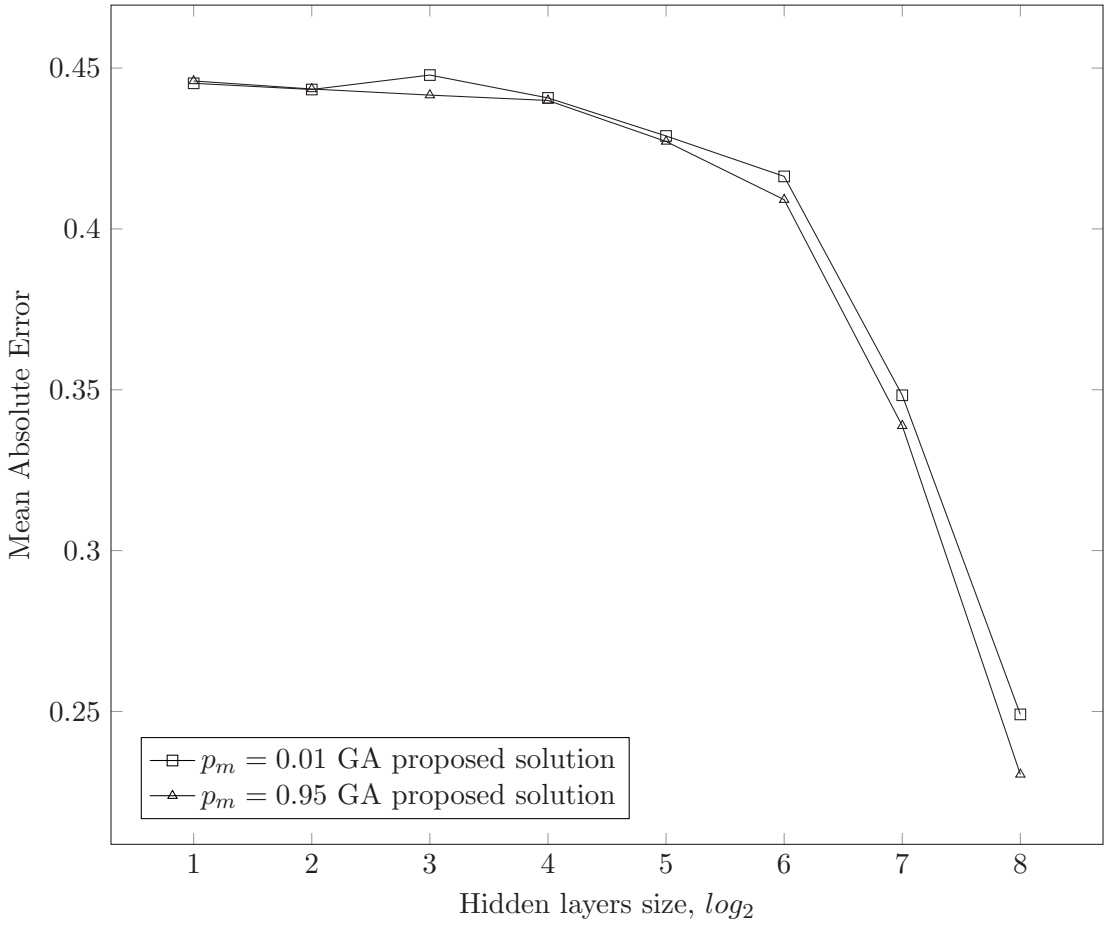


Figure 5.3.: Neuroevolution: GA proposed solutions comparison

A direct comparison of the GA proposed solutions shows that a mutation probability $p_m = 0.95$ lowers the prediction error, relative to $p_m = 0.01$.

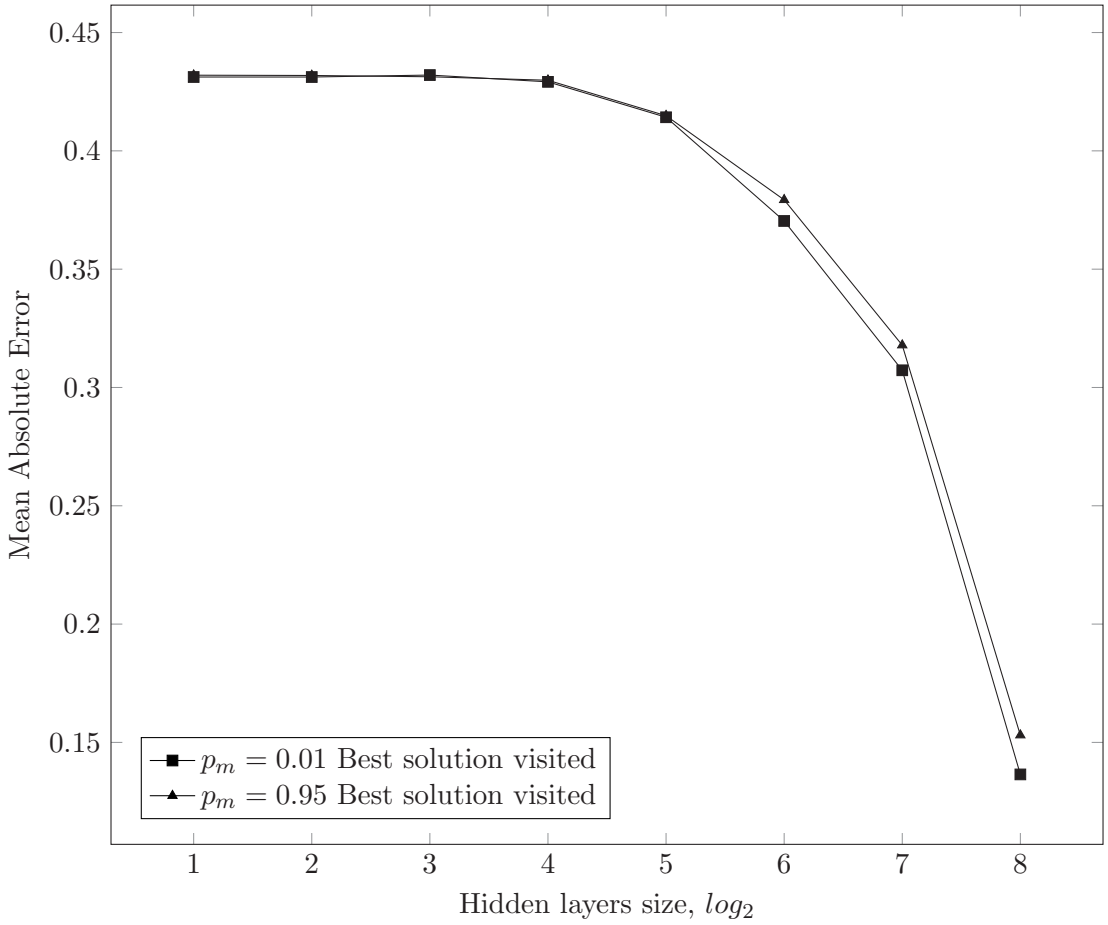


Figure 5.4.: Neuroevolution: Best solutions visited comparison

The improvement of the proposed solution comes at the cost of worsening of the best solution visited.

## 5.5. Conclusions

The experimental results (Table 5.1) show that using high-probability mutation improves the solution the GA proposes and lowers the empirically-found overfit (the difference between proposed and best visited solution).

The initial assumption was that high-probability mutation, in an adaptive avoidance of plateaus and premature convergence, would guide the Genetic Algorithm away from overfit. This experiment, aimed at testing that assumption, does not contradict it.

There are some limitations - high-probability mutation does not eliminate overfit, merely lowers it. Furthermore, while this overfit reduction results in solution improvements, it also worsens the quality of best visited solutions. This limits both the ability to draw on best visited solutions as a resource, and our ability to directly, empirically, measure overfit in high-probability mutation GAs.

Finally, the proposed approach does not cause the proposed solution to coincide with the best visited solution; due to this, it is a partial success, and bears improvement.

### 5.5.1. Practical considerations

The Neuroevolution approach described is, at the time of writing, not suitable for practical use. The best predictions are made by the largest ANNs, and those take a long time to evolve (in the above experiment, a single $2^8$-sized ANN algorithm run took approximately 5 weeks).

However, the algorithm implementation is not optimised or streamlined for a series of reasons: a simpler implementation is easier to understand, monitor and correct; larger sample sizes are needed to assess its behaviour in an experimental setting; the behaviour of the algorithm was not as well-understood before the experiment as it is after the experiment.

The use of high-probability mutation did not increase the computational resources required by the algorithm in a significant way, since it translates in an increase of genome bit-flips performed.

In order to provide faster results, both the algorithm and its manner of use can be streamlined in a number of ways:

- Depending on its use, the algorithm may be run once, to provide a single prediction, instead of the multiple runs needed to produce a sample. Since, in real-world prediction situations, the final snapshot of the data set is not known, there is no certain way of comparing prediction qualities, and multiple runs provide a less clear improvement. Instead of directly improving the prediction, multiple runs can be made to eliminate outliers.
- The GA population size, number of generations and stop conditions could be modified to reduce the number of evaluations performed. Since decoding the genome, assembling it into an ANN, and evaluation the network in the feed-forward phase are, computationally, the costliest phases of the algorithm, a reduction in the number of evaluations would translate

in an almost-linear reduction in computational resources needed and time elapsed for an algorithm run.

- Caching evaluations can also increase algorithm speed. Considering the large size of the ANNs involved, most are likely to change from generation to generation, even for very low mutation probabilities. Caching individual neuron evaluations, however, is more likely to save time. The use of high-probability mutation requires, for the simplest result-caching approaches, a doubling of the cache size used: if, for low-probability mutation, it might suffice to cache the evaluation result for the previous generation, high-probability mutation returns genomes to similar states each other generation.

- Since backpropagation is not used, there is no need for the perceptron activation function to be differentiable. Simpler, easier-to-compute functions can be used instead. However, this could impact the performance of the method, and further experiments would need to be performed.

- Genetic Algorithm evaluation is inherently parallel. The algorithm could be split in a number of threads, each run, in parallel, for an increase in speed. This approach limits the number of threads to the number of genomes in the population. That limit can be overcome, by evaluating perceptrons or groups of perceptrons in separate threads, on CPUs or GPUs [24].

# 6. Conclusions

Based upon the contents of this work, high-probability mutation can be characterised, both as an operator and as a method.

In the context of the full range of values for applying the mutation operator, effective mutation (Fig.3.1) provides a basic measure of the entropy introduced in the population by way of the mutation operator. There are high-entropy regions, and low entropy-regions; the Genetic Algorithm behaves similar to a Random Search Algorithm hybrid in the high-entropy regions, forgoing evolution.

Evolution is possible only if the mutation operator allows enough stability for the information contained in the genomes. While the precise threshold between feasible evolution and random search is sensitive to many factors (Chap. 4), the phase transition between the two does not contradict the conclusions derived from the concept of effective mutation. High values for effective mutation correspond to random search and no stable Consensus Sequences; low values for effective mutation allow for stable evolution.

Low-probability mutation lies in one low effective mutation area; high-probability mutation lies in the other. Both permit the formation of stable Consensus Sequences, and the occurrence of evolution.

Having analysed a sample of the full range of mutation probabilities (Chap. 3), characteristics of low- and high-probability mutation subsets, and having directly determined where Error Thresholds form, an account for the whole range of probability values of the mutation operator can be given:

- For $p_m$ values at or very close to 0, there is not enough exploration in the Genetic Algorithm. By merely exploiting the same solutions through recombination, finding global optima is unlikely, and the GA performs poorly.

- For non-0 mutation probabilities bellow the Error Threshold, the mutation operators allows for stable evolution, while the errors it introduces in genomes push it to explore new solution candidates.
- For mutation values in a wide range centred around $p_m = 0.5$, where stable Consensus Sequences cannot form, evolution does not take place. Instead, a hybrid Random Search Algorithm is created. There are two ways of identifying this range: first is by computing the entropy mutation imparts to the population, not in a generation, but across multiple ones; second, by identifying the central area on the mutation probability range of values that is delimited by Error Thresholds.
- As mutation probability increases beyond a certain value, the cumulative errors it imparts on the population start decreasing, eventually below the Error Threshold beyond which evolution can happen again. This is the area of high-probability mutation.
- Very close to or at $p_m = 1$ , the rate of errors induced in the population is insufficient to drive exploration. As such, the algorithm returns to exploiting the same areas of the landscape, with generally poor results.
- Small variations in the probability of applying the mutation operator can still lead to large changes, and problem-specific particularities are expected. The $p_m$ values used, while aimed at sampling the whole $[0, 1]$ range, do not and cannot cover the whole range; the assumption underlying the sampling is that very similar mutation probabilities will lead to similar Genetic Algorithm behaviour. In the context of the huge number of possible $p_m$ values, it is likely a large number of exceptions exist.

High-probability mutation behaves differently from low-probability mutation. By creating an approximate dual, binary-negation representation for each genome, very fit genomes are prevented from replicating until they fill the population. Instead, in the "dual" generations, other genomes are given the chance to reproduce. For most function landscapes, there are no special links between a representation and its dual; through mere statistics, it is likely that, if a genome is close to the optimum, its Boolean negation will be farther away (assuming the function is not dense with true optima - otherwise, the optimum is simple to find). By taking this step back, high-probability mutation prevents the population from being locked into an area of the

landscape - it prevents population convergence.

When navigating a plateau, due to the low fitness differences between genomes, differential selection is slow to follow promising leads: plateaus are slow to navigate. By working in the approximate dual representation, high-probability mutation offers "shortcuts", ways for the population to explore other areas of the landscape, less by the generally good fitness of the plateau. Because each dual representation is different - due to the uniformly random nature of Boolean negation mutation - these shortcuts are also random; while they don't take advantage of function landscape characteristics, other than fitness differences common to difficult-to-solve functions, they are also less likely to be deceived by particular landscape features.

The same features make high-probability mutation slow to converge to good optima - there is no general way to know, ahead of time, if an attraction basin leads to a local or global optimum. If there is little danger for premature convergence, or if plateaus are not a significant feature of the problem being solved, using high-probability mutation will delay convergence towards the global optimum, and lower Genetic Algorithm performance. These characteristics can be explained in the context of the No Free Lunch Theorem[45]: while, overall, both methods could have similar performances, the use of problem and instance-specific knowledge when deciding which method to apply on which instance allows us to avoid the comparative disadvantages of both methods, and exploit their advantages.

Thus, a rule-of-thumb regarding the use of high-probability mutation can be formulated: If the function landscape is known or suspected to contain plateaus, or if the Genetic Algorithm struggles with premature convergence, high-probability mutation should be used.

This rule was put to the test in Chap. 5, where high-probability mutation lowered the overfit incurred in a Neuroevolution algorithm.

From a practical perspective, the most important aspect of high-probability mutation is its ease-of-use: it consists of simply choosing a mutation operator probability $\approx 0.95$. In the Simple Genetic Algorithm case and its close derivatives, there is no need to perform other algorithm changes. For existing algorithmic implementation, this generally means there is no additional development cost, source code modification or, for closed-source or antiquated implementations, recompilation.

This also means that the computational costs associated with high-probability mutation are very small. In most Genetic Algorithms, the time devoted to mutation is small compared to the

costlier operations, like genome evaluation.

## 6.1. Comparison to similar methods

There are a few Genetic Algorithm variants that are similar to high-probability mutation, either in approach or results.

### 6.1.1. Hypermutation

Hypermutation[41] and Triggered Hypermutation[29] involve increasing both mutation and effective mutation rates past the Error Threshold, sometimes raising them to $p_m = 0.5$. Triggered Hypermutation aims at resetting evolution if the current direction it has taken is judged suboptimal.

While being fundamentally different methods - high-probability mutation is evolutionary, Hypermutation is random - they have a partial overlap, in attempting to solve premature convergence.

High-probability mutation lets selection pressure and the dynamic sub-population replication rates between primary and approximate-dual representations regulate premature convergence. Triggered Hypermutation measures population stagnation and intervenes post-factum.

High-probability mutation has the comparative advantage of not needing additional algorithm complexity and preserving the information evolved within the population; it, however, impedes the way the GA approaches "straightforward" optima. If triggered at appropriate times (and the detection of these times is a problem within itself), Hypermutation has the possibility of intervening just when needed.

The two methods can be combined in multiple ways; first, Hypermutation can be used to supplement high-probability mutation - by lowering the mutation rate from $\approx 0.95$ towards 0.5, effective mutation rates increase.

Second, high-probability mutation can be triggered the same way Hypermutation is; the GA can be allowed to pursue its default strategy and, when premature convergence is detected, high-probability mutation can be activated, without the scrambling of the information contained in the population, and without interrupting evolution. Any short-burst (i.e. low number of generations) of high-probability mutation needs to return the population in its primary encoding. The simplest

way of doing that for Genetic Algorithms is to have an even number of high-probability mutation generations.

### 6.1.2. Primal-dual chromosomes

Primal-dual Chromosomes[46] retain their precise Boolean-negated representation, and are evaluated as the maximum fitness of either. This method has good performance on the GA Royal Road Function.

Primal-dual Chromosomes, compared to high-probability mutation, differ in a number of ways:

- The Primal-dual method evaluates a chromosome as the best of either its primary or Boolean-negated encoding. In contrast, high-probability mutation selects a genome according to both, in successive generation. During plateaus or premature convergence, the two methods function differently: Primal-dual Chromosomes have no mechanism to push them away from the local optimum.
- The Primal-dual Chromosomes algorithm is suitable for bit-block functions with simple optimum solution representations. The GA Royal Road and GA Trap Functions, in particular, highlight this advantage. However, the problem-related knowledge inserted into the algorithm in these cases is very high, stopping short of the solution representation: the Primal-dual Chromosome seems custom-made to solve them. High-probability mutation seems to have a wider range of applicability.
- While, for simple bit-block functions, both the primal and dual chromosomes can be evaluated in the same step[46], in the general case the method requires twice the number of evaluations.
- High-probability mutation creates partial Boolean-negation representations for each genome, each dual generation. This allows the method to explore a wider range of possible representations than the single complete Boolean negation, even in the worst-case scenario of a static primary population.

### 6.1.3. Diploid Algorithms

There are genetic algorithms which store mostly-similar genetic encodings in two (diploid) or multiple (polyploid, also known as multiploid) separate chromosomes[18].

By defining a dominant-recessive relation between homologous genes, previous representations, which were fit in the past but are no longer, can be shielded from deletion. If the current context changes to its disfavour, the genome can attempt to express (be evaluated according to) the recessive genes.

Such Diploid Algorithms are used with non-stationary optima and function landscapes, in dynamic optimisation problems. The algorithm maintains a library of genetic information safe from extinction, by having at least one fit set of genes.

While the approximate alternate representations and continuous evolution are common traits between Diploid Genetic Algorithms and high-probability mutation GAs, they differ in the way they function, the complexity of each, and the preferred applicability domain.

The most relevant difference is that high-probability mutation does not shield suboptimal genomes and genetic sequences from extinction, in the long term. Rather, by subjecting genomes to selection both in their primary and Boolean-negated representations, it delays rapid convergence and thus, rapid deletion of suboptimal sequences. The protective nature of high-probability mutation applies only in the short term.

## 6.2. Future work

### 6.2.1. Non-binary integer encodings

High-probability mutation is not necessarily bound to binary-encoded Genetic Algorithms. It requires that the population be returned to its primary encoding in a small number of generations - and this can be achieved if the mutation paths a way for this synchronised return.

If each locus had 4 possible alleles, an incremental mutation could return the population to the initial representation in 4 generations: $0 \to 1 \to 2 \to 3 \to 0$. In fact, any mutation able to synchronise the return to the initial representation would do: $0 \to 1 \to 3 \to 2 \to 0$.

This could apply for integer or fixed-point number representations, as long as the number of possible alleles in each locus is kept low.

Since the effective mutation rate over multiple generations increases, the mutation probability will likely be high, higher than the studied $\approx 0.95$.

### 6.2.2. Approximate Consensus Sequences

In some of the Consensus Sequence Plots investigated, small and temporary Consensus Sequence fluctuations decreased the measured Error Threshold (e.g. Fig. A.9). In other cases, the significant bits of the genome are fixed early on, and the less-significant bits fluctuate afterwards (Fig. A.13-A.36). In both these, there is evolved information in the population that is not disrupted by mutation; by modifying what precisely constitutes a Consensus Sequence change, and thus changing where the Error Threshold is placed, we could more accurately reflect the transition between functioning and disrupted mutation.

Additionally, in the small-scale test performed leading to the experiments described in Chap. 4, non-elitist generational GA was found to have very low Error Thresholds. This was caused, in most cases, by small, temporary fluctuation. A less strict definition for Consensus Sequence change would allow identifying Critical Mutation Rates for them, as well.

For example, two Consensus Sequences that differ for less than 5% of their loci could be considered similar. More-significant loci (in the simple numeric sense) could weigh in more in the comparison, while less-significant loci - less.

This proposal is being made because of the experimental conclusions drawn in Chap. 4; however, there is evidence in molecular biology that relaxed Error Thresholds exist[22] in biological evolution.

### 6.2.3. Consensus Sequences on multiple quasispecies

Consensus Sequences display the majority allele for each locus, across the whole population. One of the inherent assumptions is that such a Consensus Sequence is representative for the whole population.

In a single population, multiple sub-populations can exist, each occupying a different area of the landscape, and interacting differently with the operators and other sub-populations. In molecular biology, each of these sub-population is called a quasispecies, and it is formed by all similar genomes within that population.

By clustering similar genomes together and deriving Consensus Sequences for each group, the behaviour of each quasispecies can be closely monitored. Thus, top-level Consensus Sequence changes that are, in fact, one quasispecies becoming more populous than the others, could be prevented from lowering the Critical Mutation rate.

# Appendices

# A. Consensus Plots

This appendix contains Consensus Sequence Plots, used in finding Error Thresholds, as described in Chapter 4.

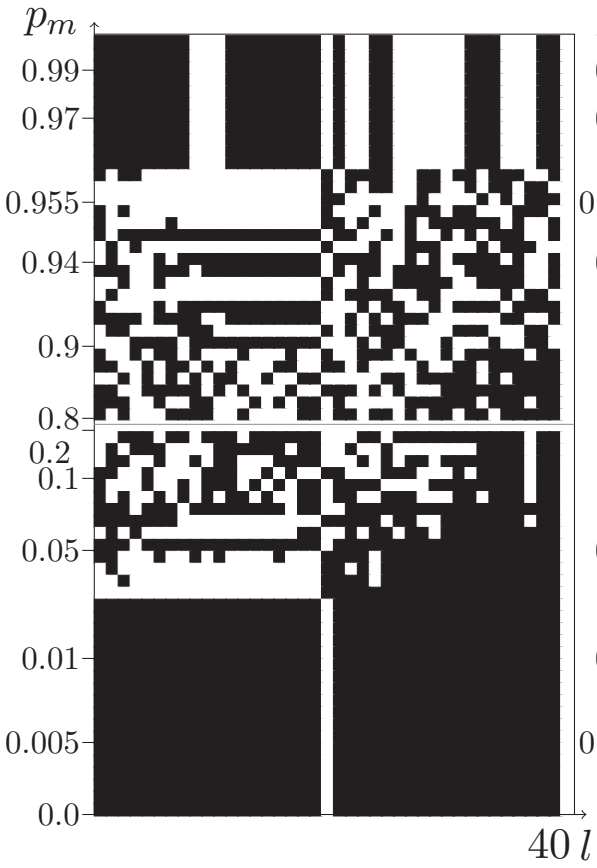


Figure A.1.: Consensus Plot: Rastrigin's Function, pop 10, destructive cx

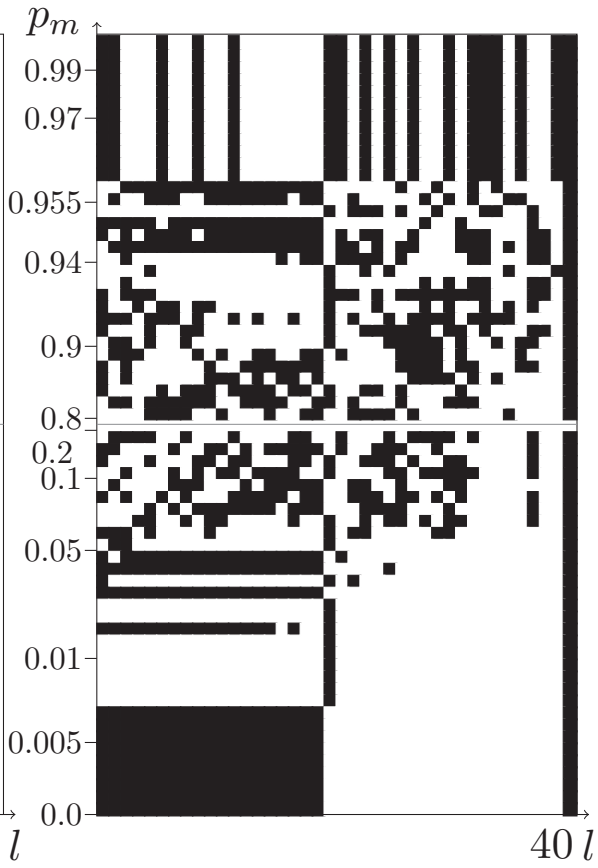


Figure A.2.: Consensus Plot: Rastrigin's Function, pop 10, nondestructive cx

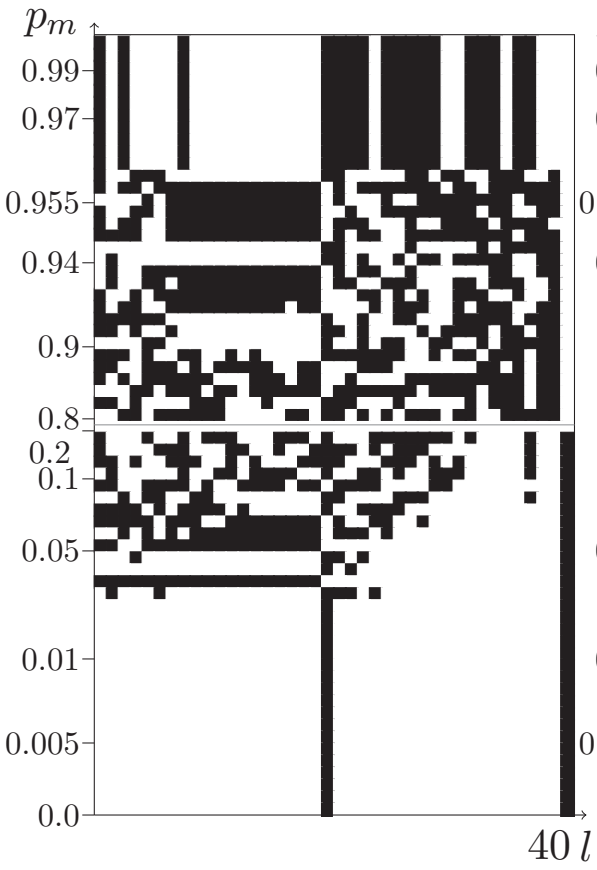


Figure A.3.: Consensus Plot: Rastrigin's Function, pop 25, destructive cx

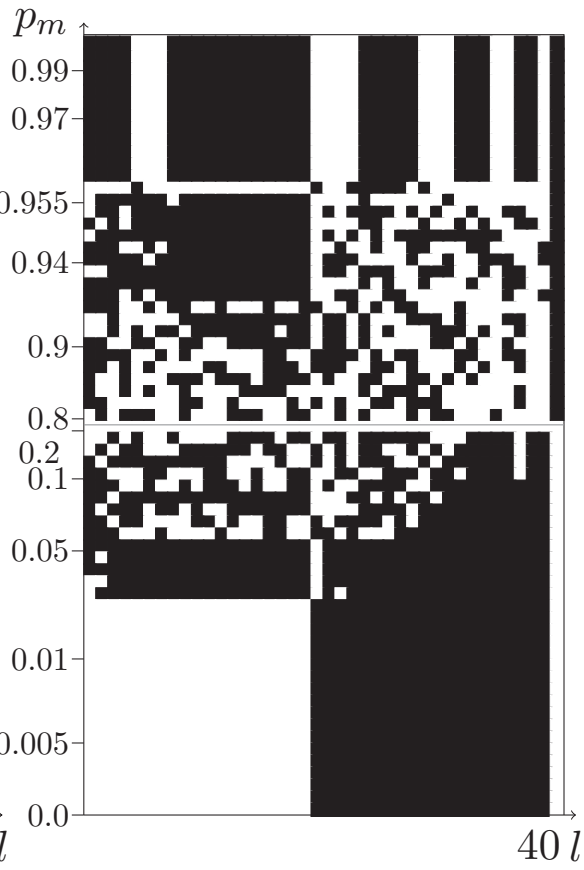


Figure A.4.: Consensus Plot: Rastrigin's Function, pop 25, nondestructive cx

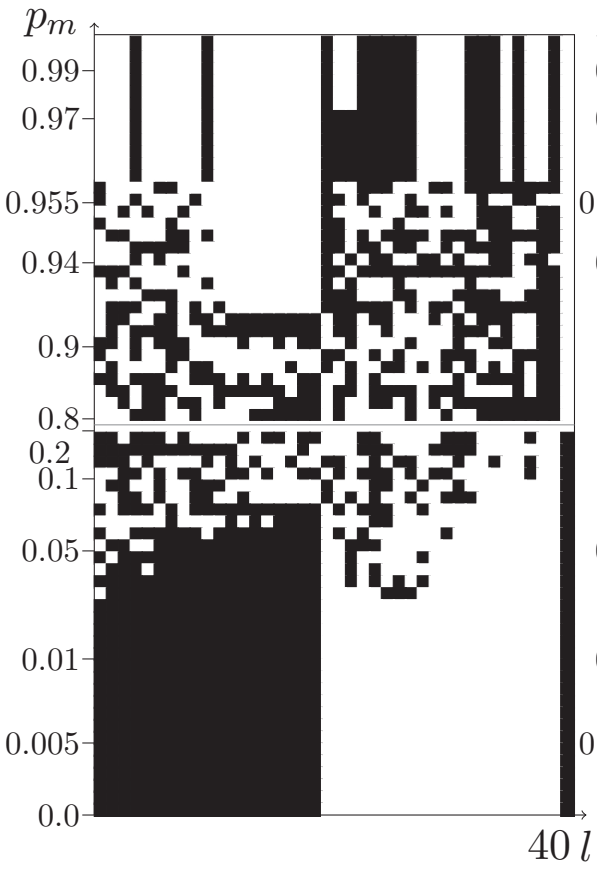


Figure A.5.: Consensus Plot: Rastrigin's Function, pop 50, destructive cx

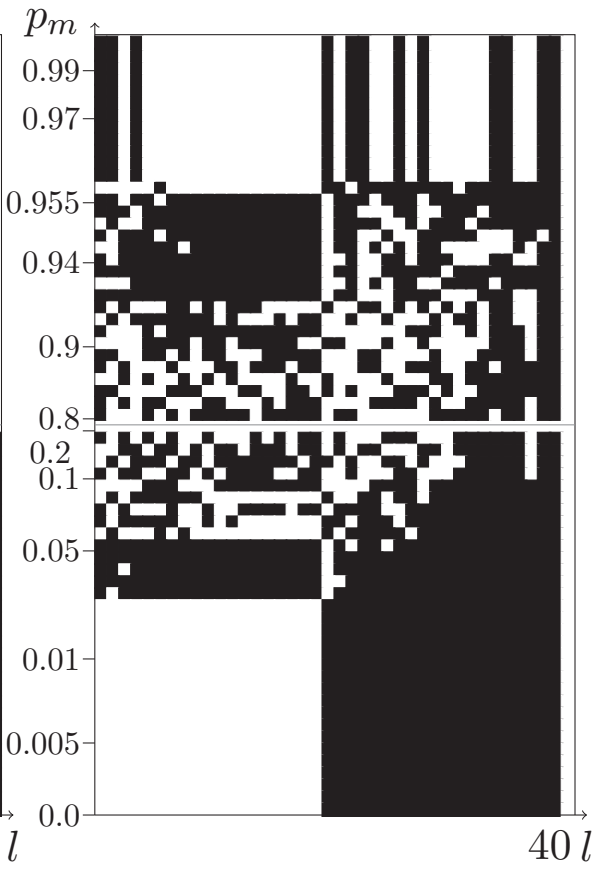


Figure A.6.: Consensus Plot: Rastrigin's Function, pop 50, nondestructive cx

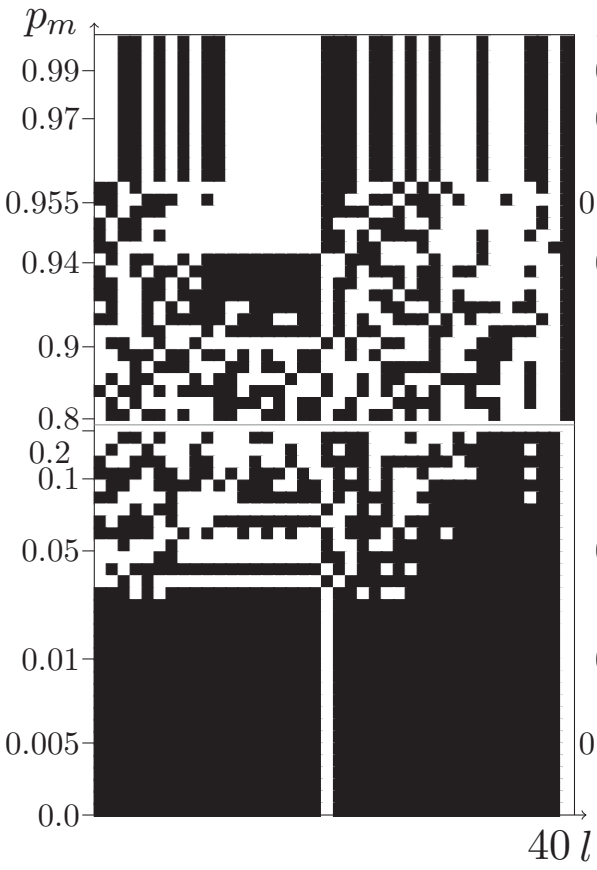


Figure A.7.: Consensus Plot: Rastrigin's Function, pop 75, destructive cx

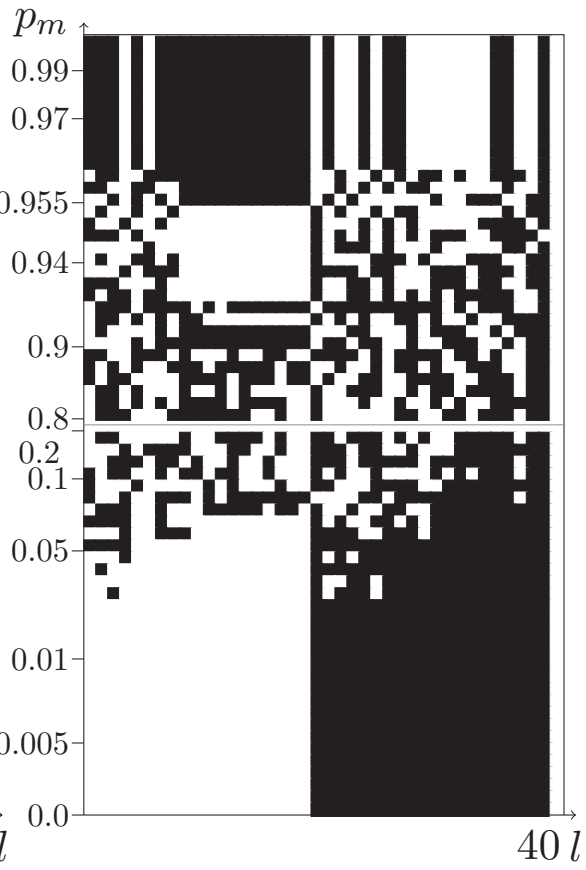


Figure A.8.: Consensus Plot: Rastrigin's Function, pop 75, nondestructive cx

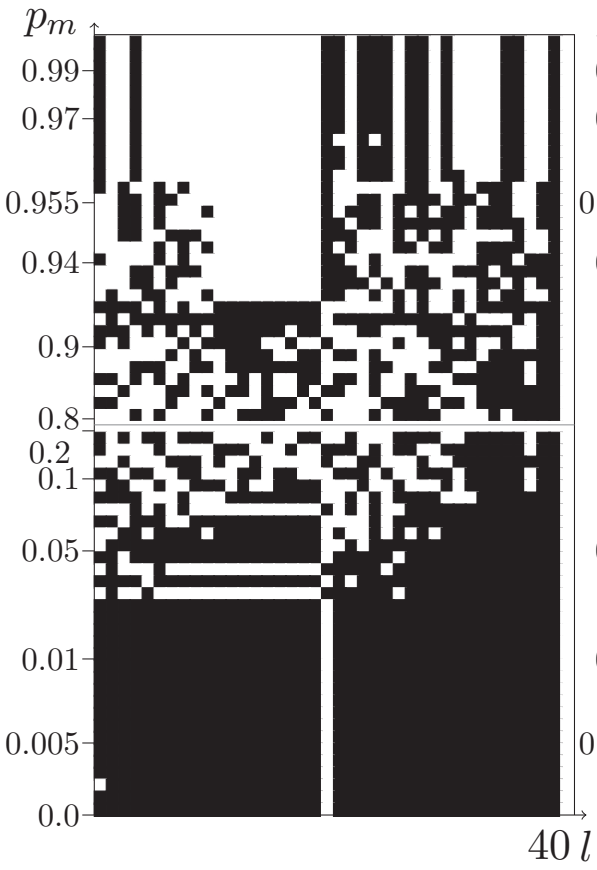


Figure A.9.: Consensus Plot: Rastrigin's Function, pop 100, destructive cx

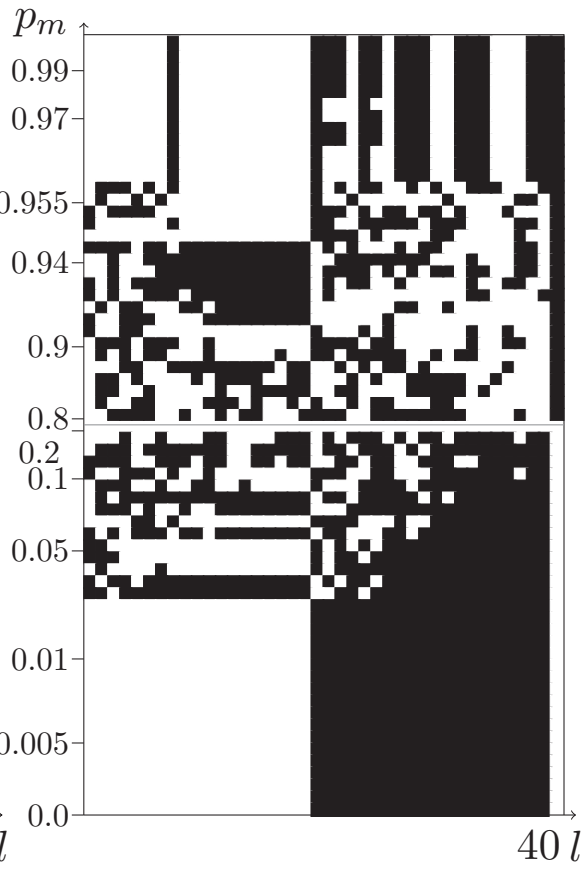


Figure A.10.: Consensus Plot: Rastrigin's Function, pop 100, nondestructive cx

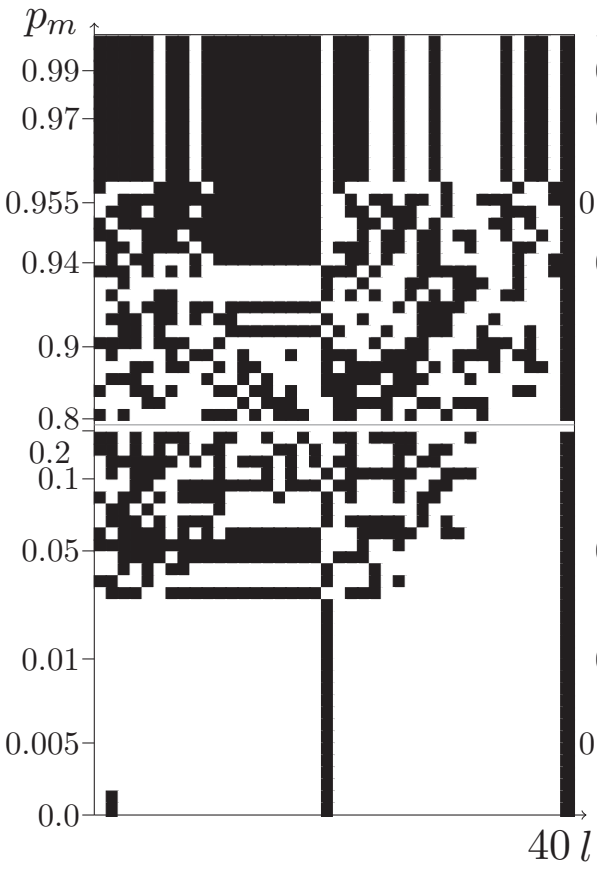


Figure A.11.: Consensus Plot: Rastrigin's Function, pop 200, destructive cx

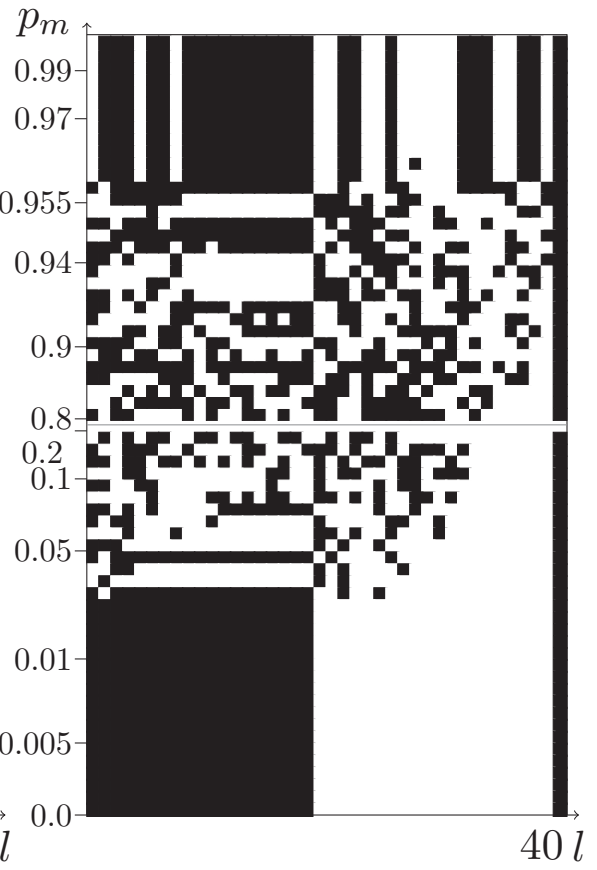


Figure A.12.: Consensus Plot: Rastrigin's Function, pop 200, nondestructive cx

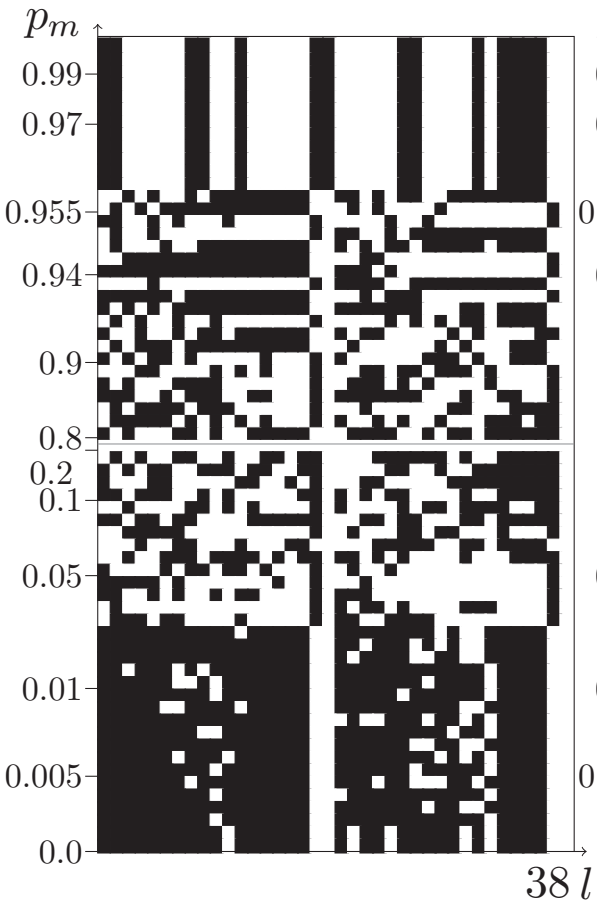


Figure A.13.: Consensus Plot: Rosenbrock's Function, pop 10, destructive cx

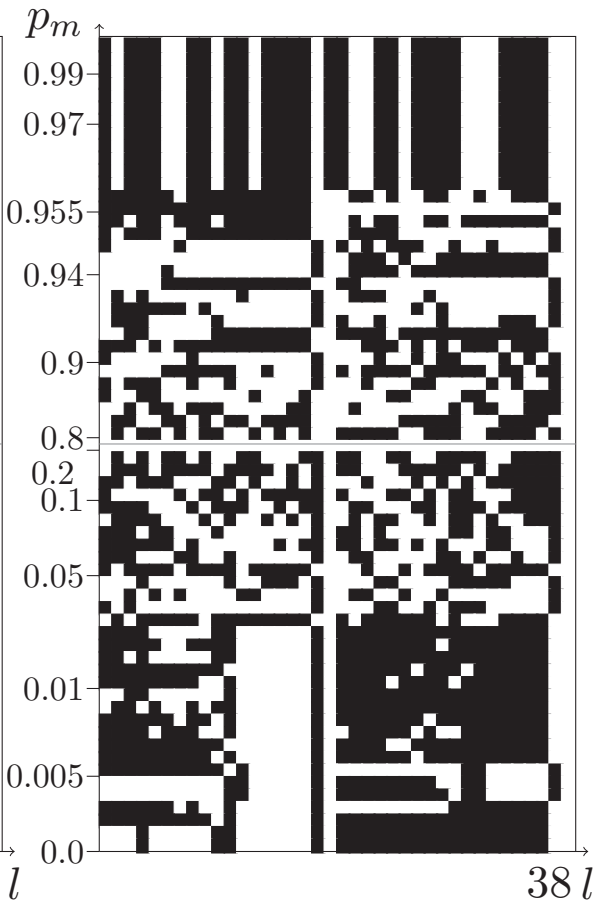


Figure A.14.: Consensus Plot: Rosenbrock's Function, pop 10, nondestructive cx

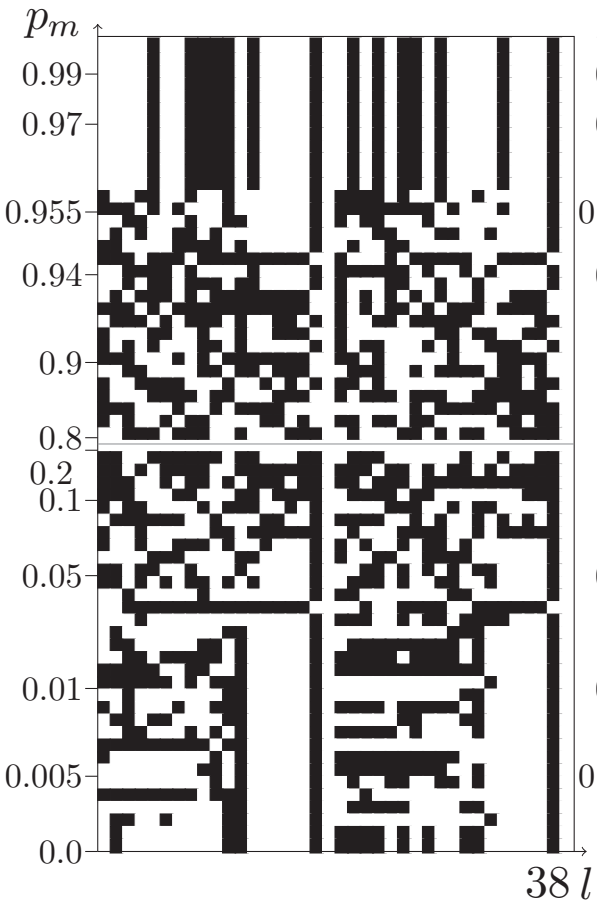


Figure A.15.: Consensus Plot: Rosenbrock's Function, pop 25, destructive cx

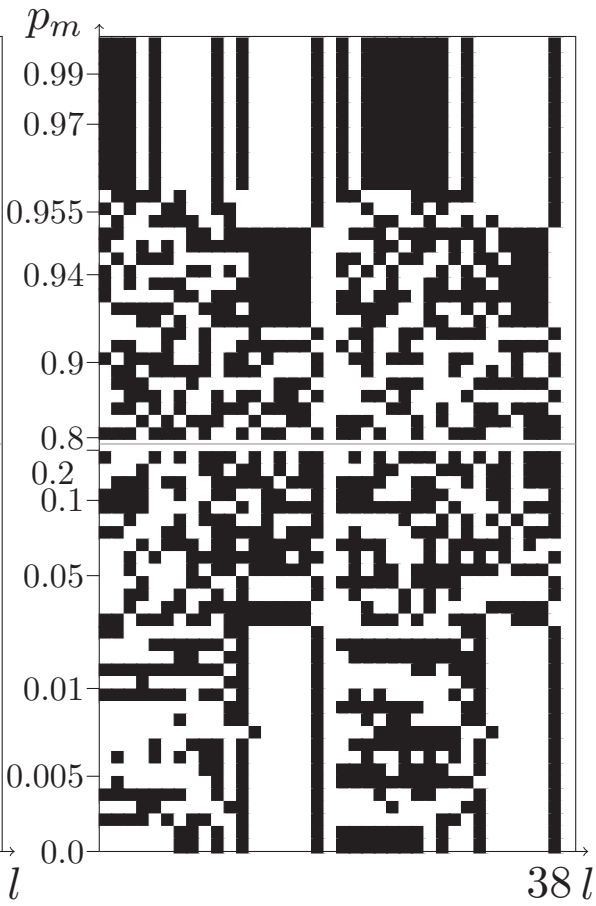


Figure A.16.: Consensus Plot: Rosenbrock's Function, pop 25, nondestructive cx

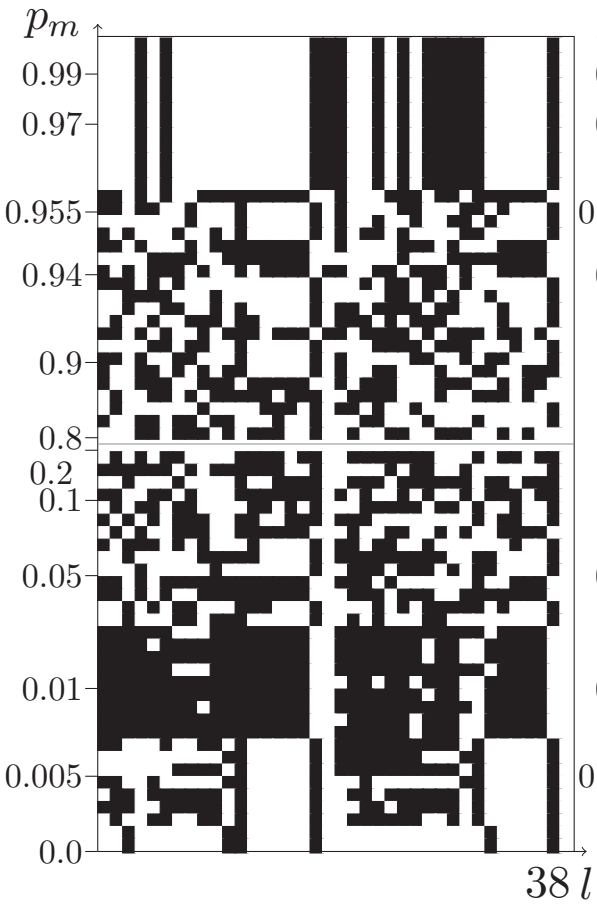


Figure A.17.: Consensus Plot: Rosenbrock's Function, pop 50, destructive cx

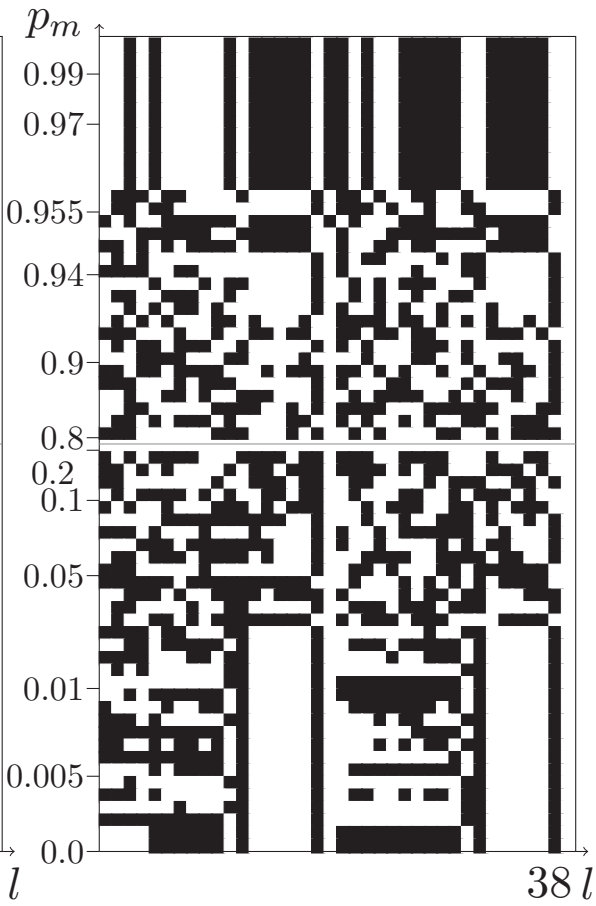


Figure A.18.: Consensus Plot: Rosenbrock's Function, pop 50, nondestructive cx

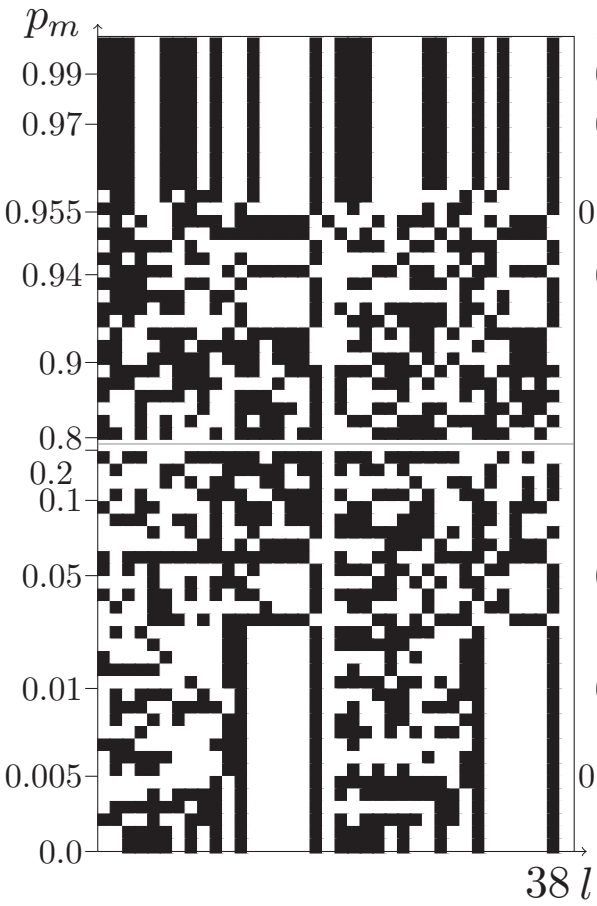


Figure A.19.: Consensus Plot: Rosenbrock's Function, pop 75, destructive cx

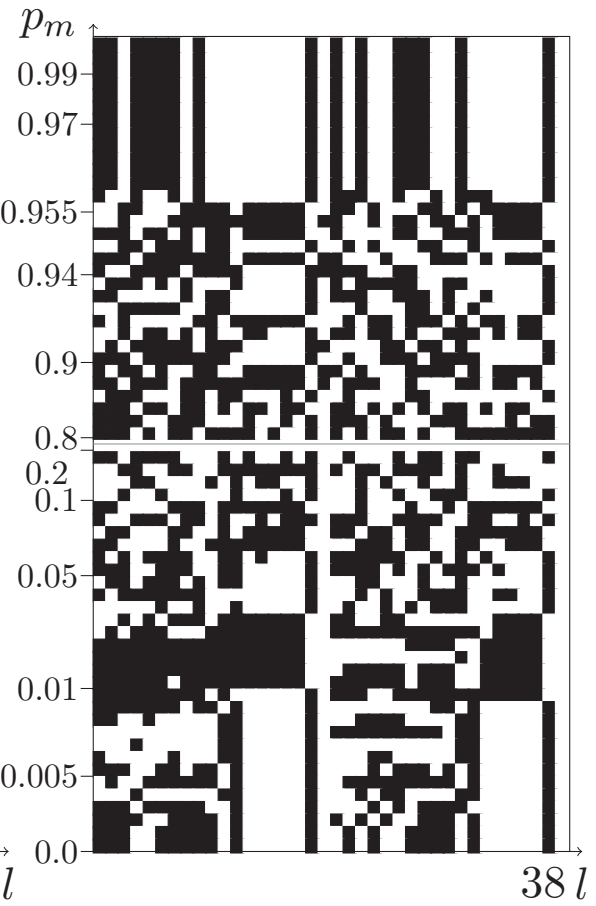


Figure A.20.: Consensus Plot: Rosenbrock's Function, pop 75, nondestructive cx

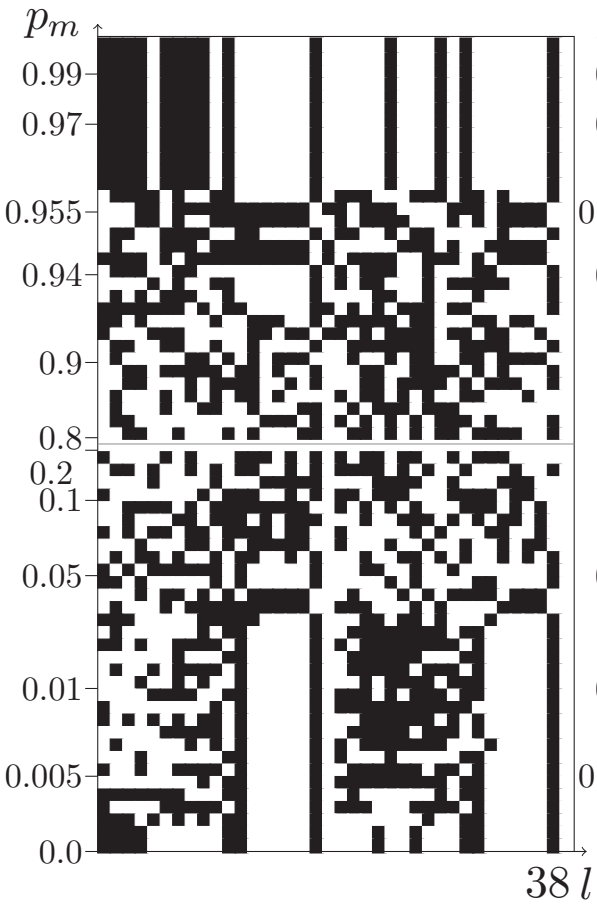


Figure A.21.: Consensus Plot: Rosenbrock's Function, pop 100, destructive cx

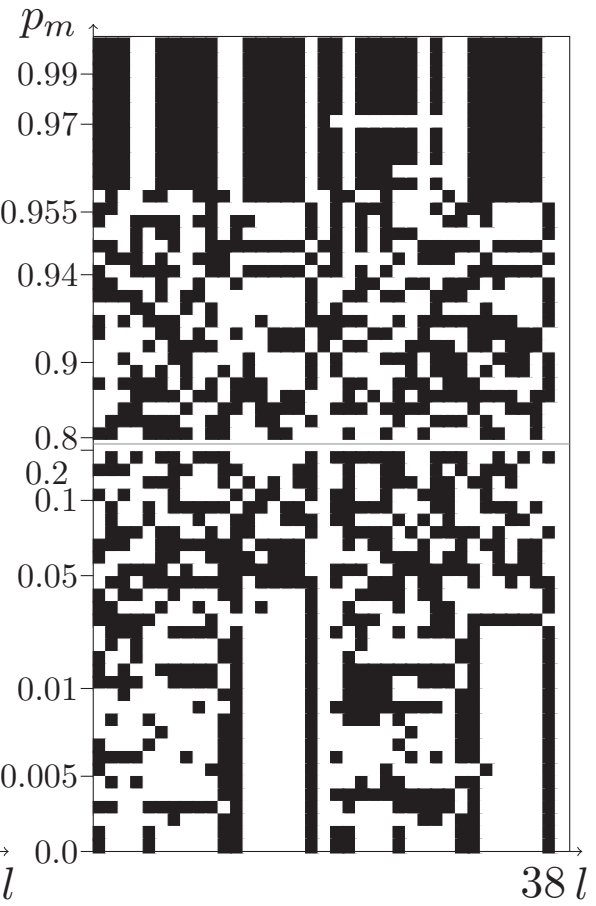


Figure A.22.: Consensus Plot: Rosenbrock's Function, pop 100, nondestructive cx

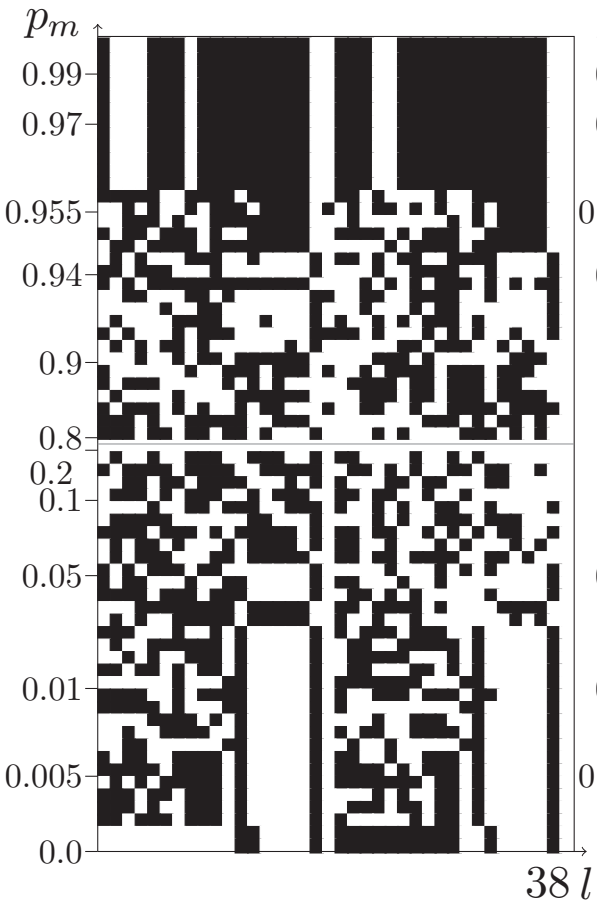


Figure A.23.: Consensus Plot: Rosenbrock's Function, pop 200, destructive cx

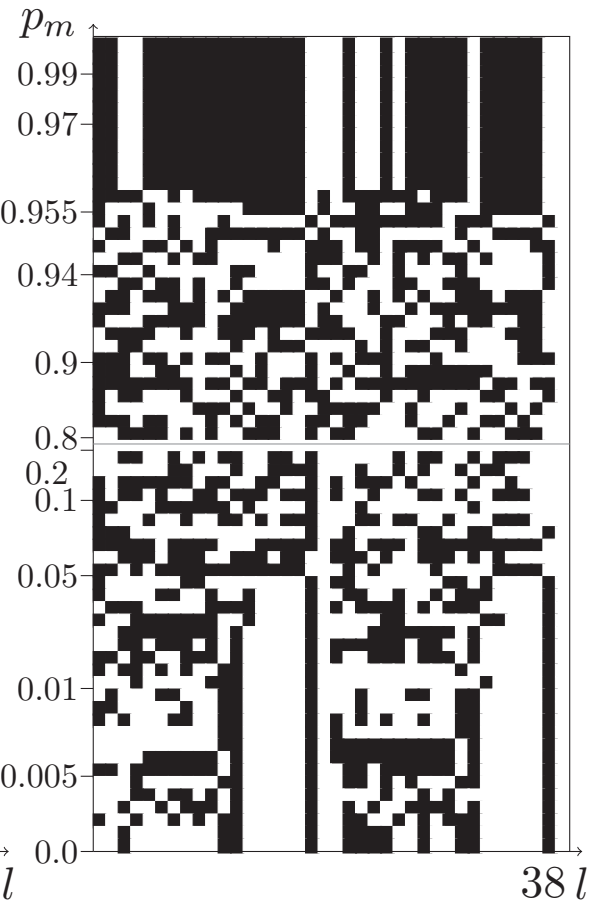


Figure A.24.: Consensus Plot: Rosenbrock's Function, pop 200, nondestructive cx

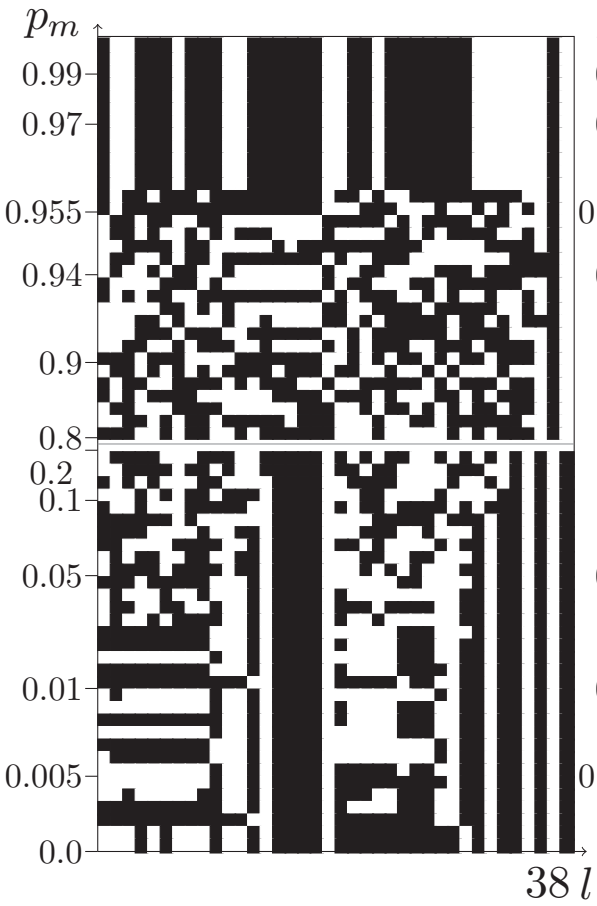


Figure A.25.: Consensus Plot: Six-Hump Camelback Function, pop 10, destructive cx

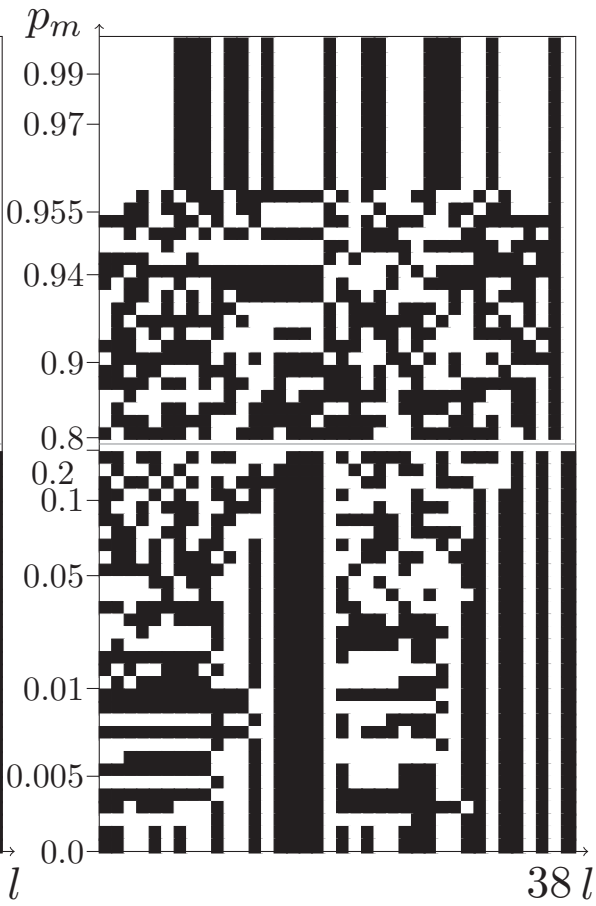


Figure A.26.: Consensus Plot: Six-Hump Camelback Function, pop 10, nondestructive cx

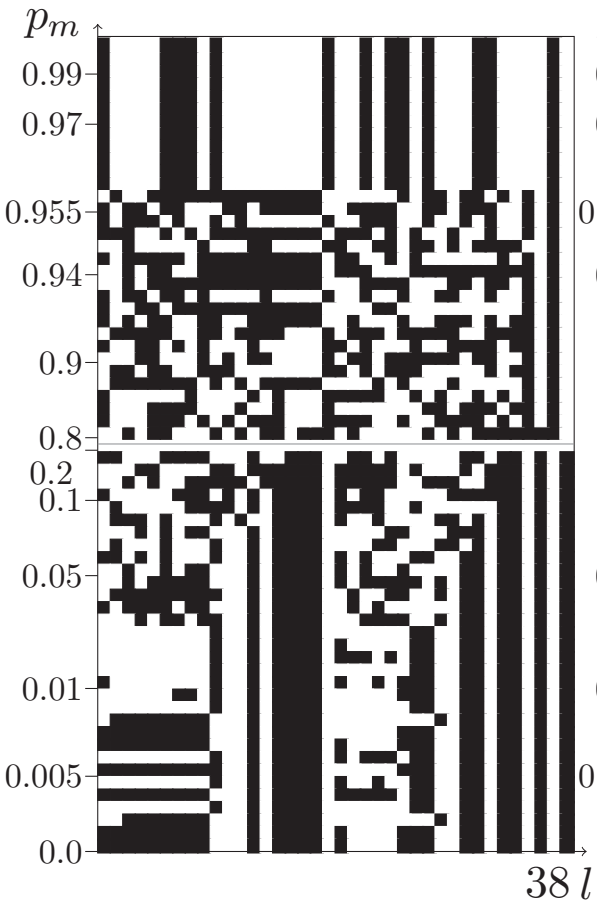


Figure A.27.: Consensus Plot: Six-Hump Camelback Function, pop 25, destructive cx

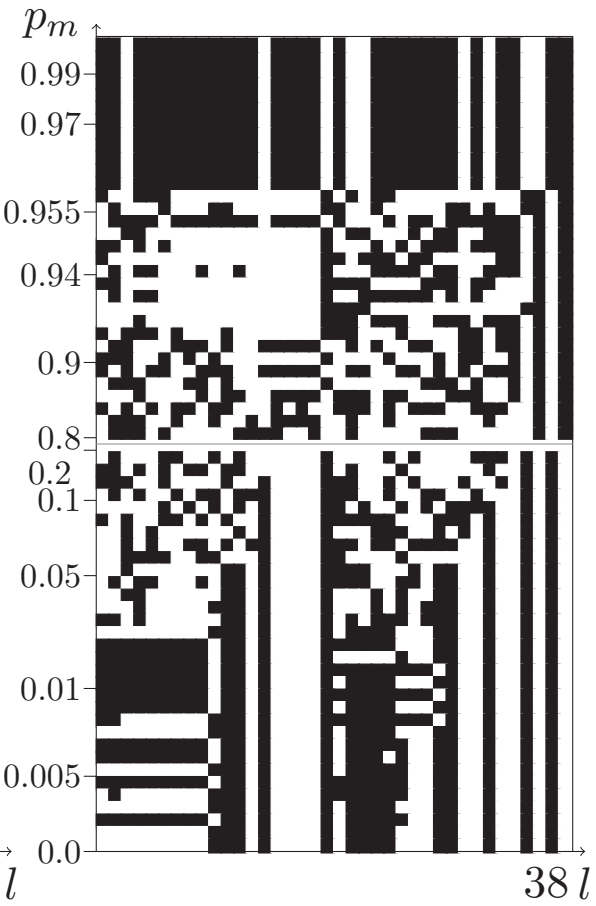


Figure A.28.: Consensus Plot: Six-Hump Camelback Function, pop 25, nondestructive cx

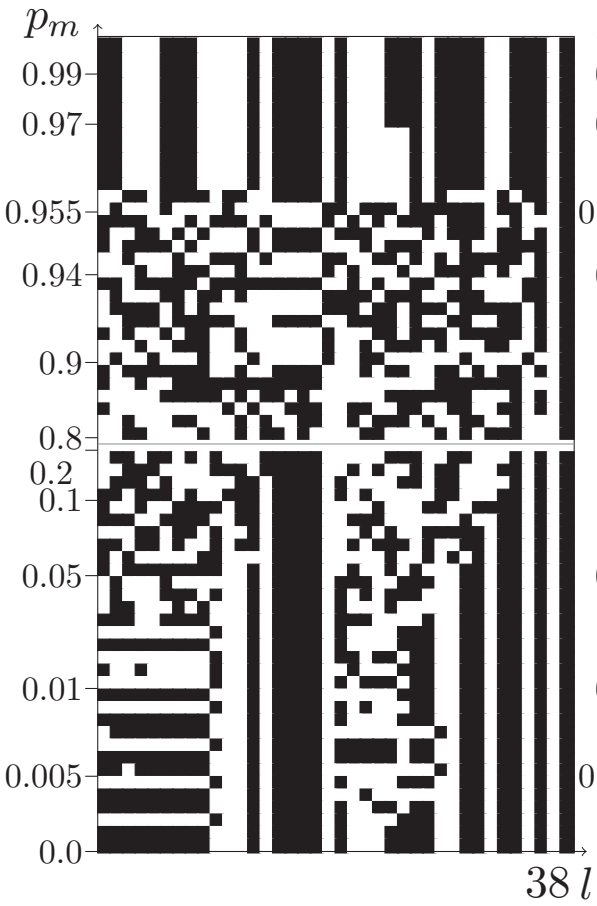


Figure A.29.: Consensus Plot: Six-Hump Camelback Function, pop 50, destructive cx

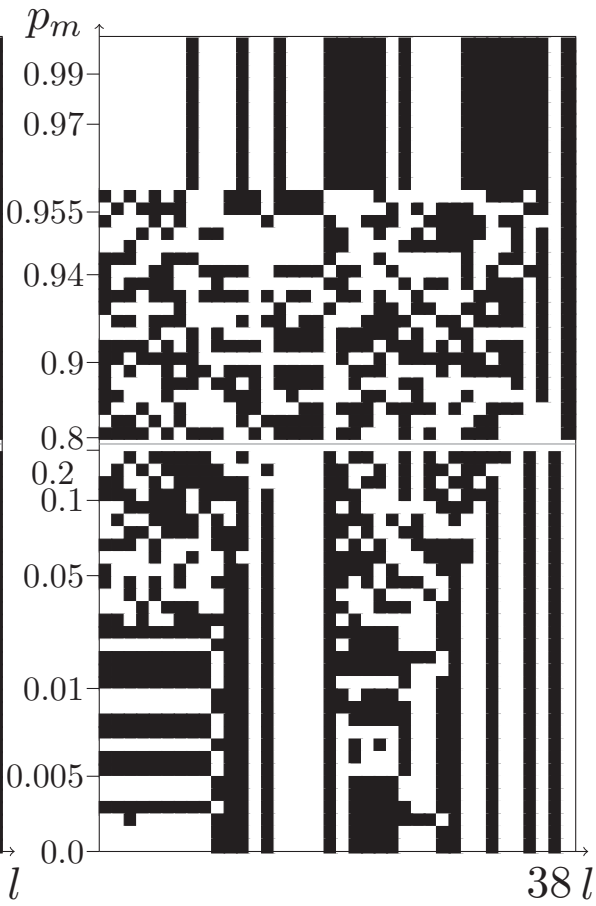


Figure A.30.: Consensus Plot: Six-Hump Camelback Function, pop 50, nondestructive cx

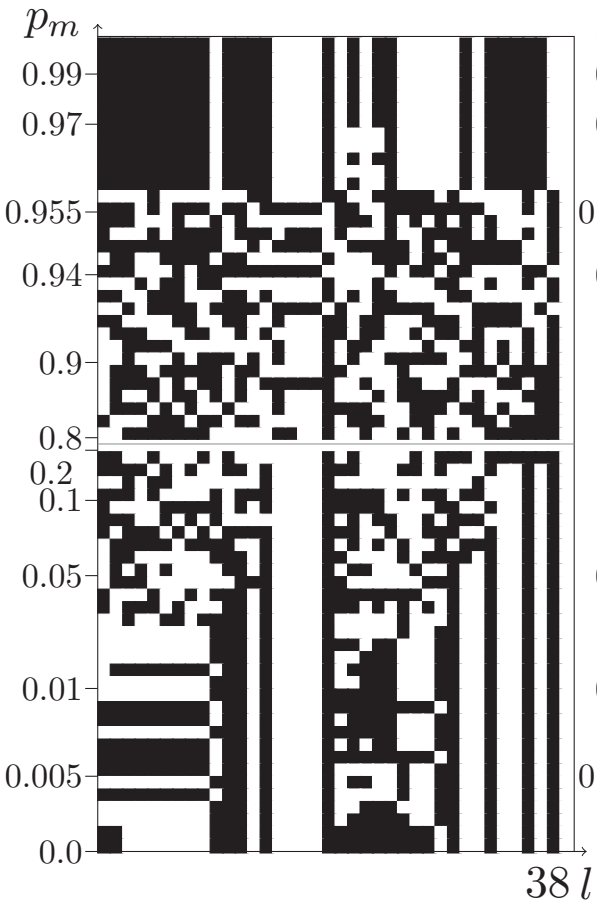


Figure A.31.: Consensus Plot: Six-Hump Camelback Function, pop 75, destructive cx

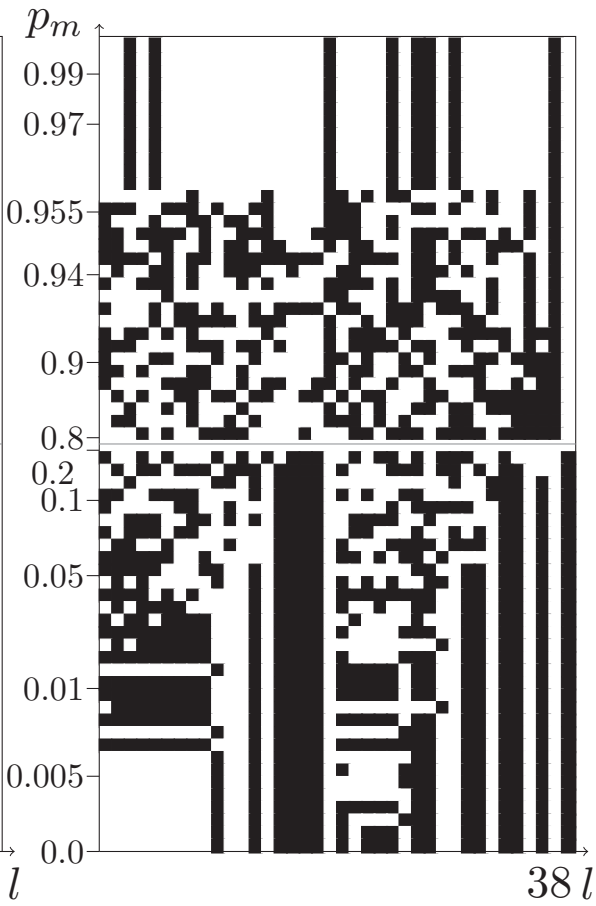


Figure A.32.: Consensus Plot: Six-Hump Camelback Function, pop 75, nondestructive cx

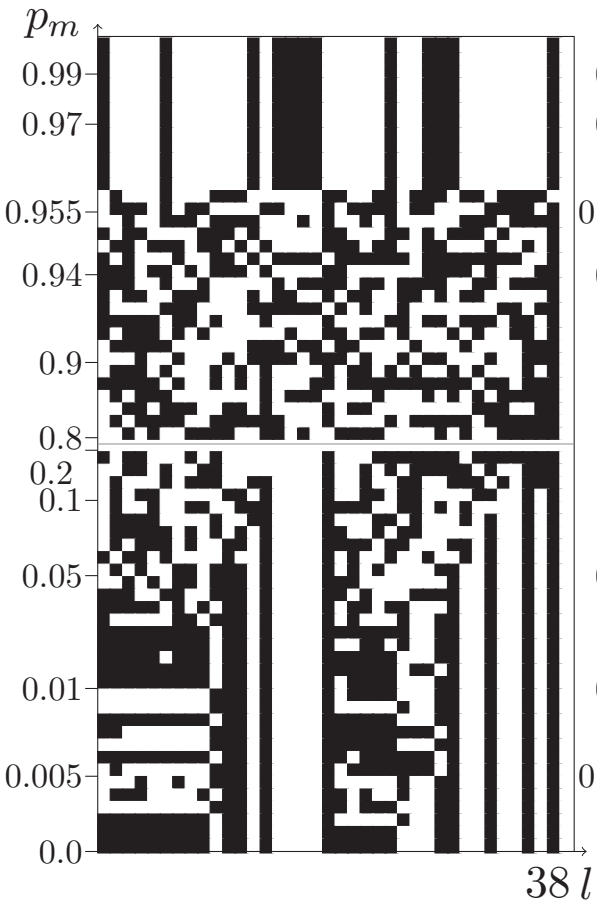


Figure A.33.: Consensus Plot: Six-Hump Camelback Function, pop 100, destructive cx

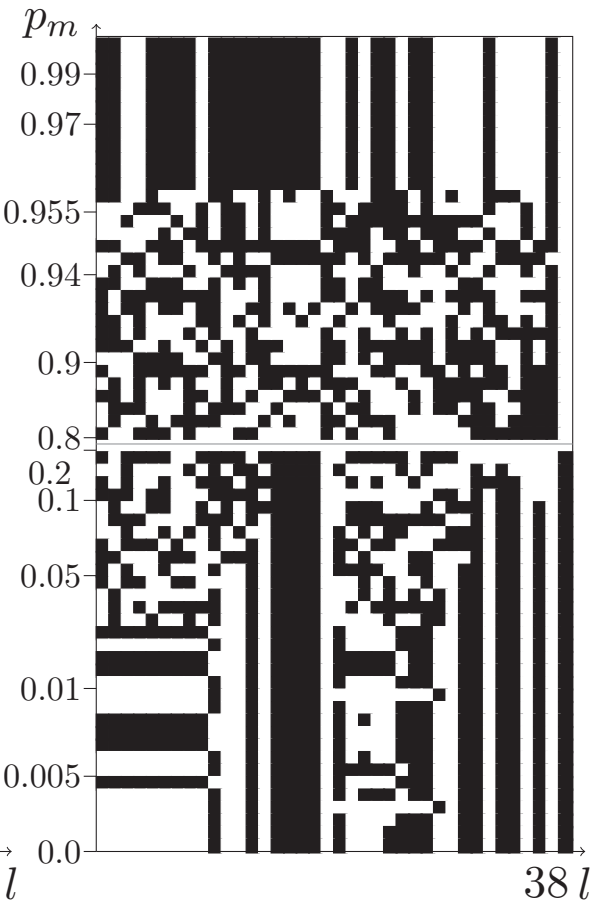


Figure A.34.: Consensus Plot: Six-Hump Camelback Function, pop 100, nondestructive cx

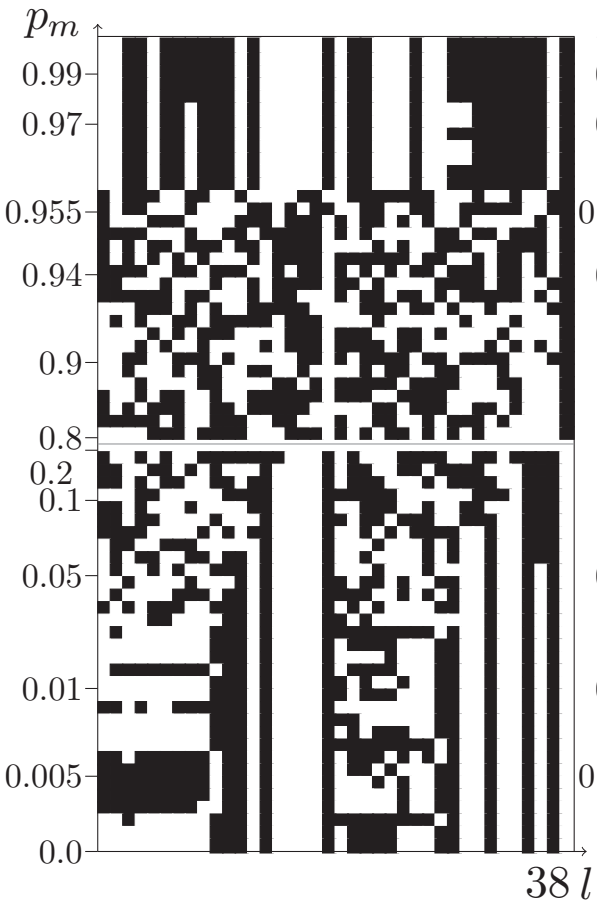


Figure A.35.: Consensus Plot: Six-Hump Camelback Function, pop 200, destructive cx

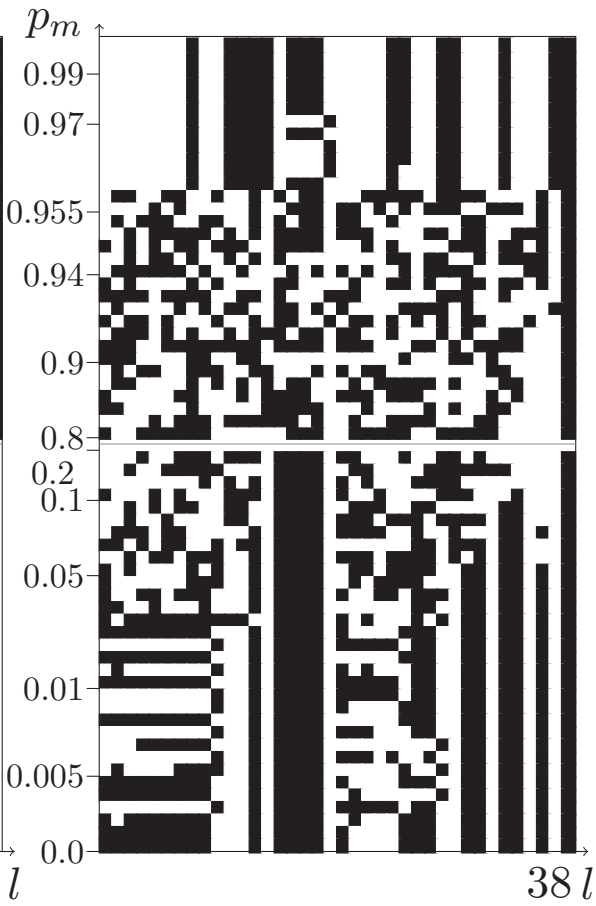


Figure A.36.: Consensus Plot: Six-Hump Camelback Function, pop 200, nondestructive cx

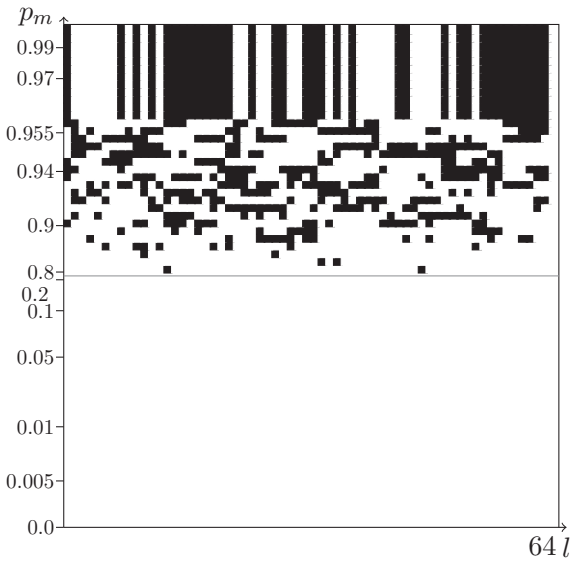


Figure A.37.: Consensus Plot: GA Royal Road Function, pop 10, destructive cx

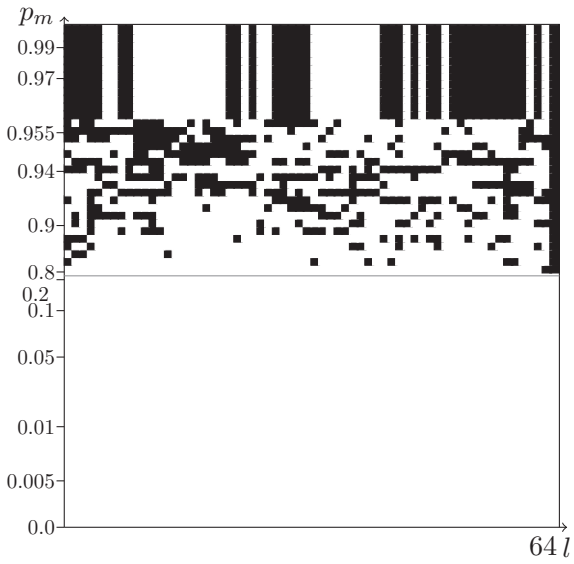


Figure A.38.: Consensus Plot: GA Royal Road Function, pop 10, nondestructive cx

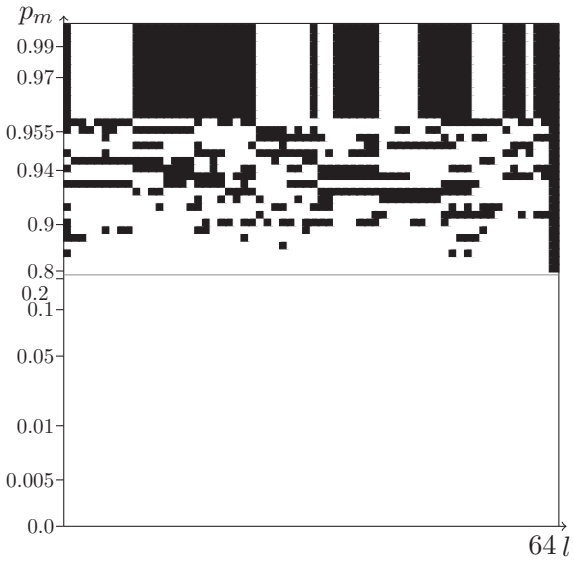


Figure A.39.: Consensus Plot: GA Royal Road Function, pop 25, destructive cx

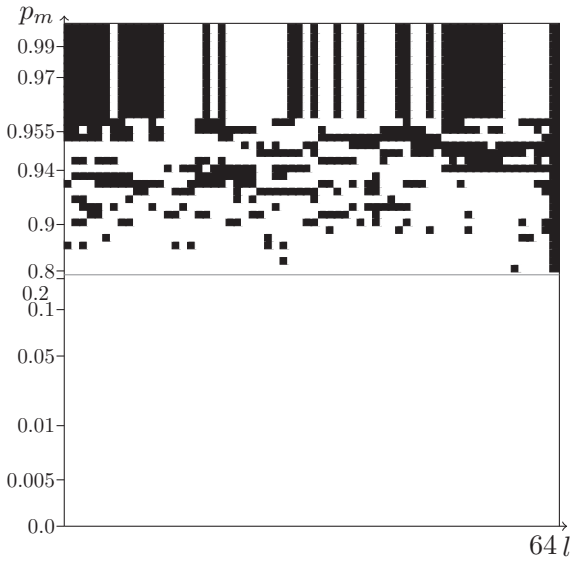


Figure A.40.: Consensus Plot: GA Royal Road Function, pop 25, nondestructive cx

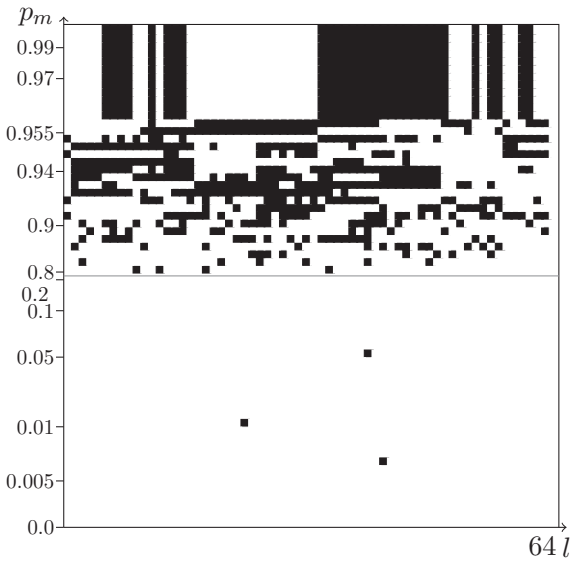


Figure A.41.: Consensus Plot: GA Royal Road Function, pop 50, destructive cx

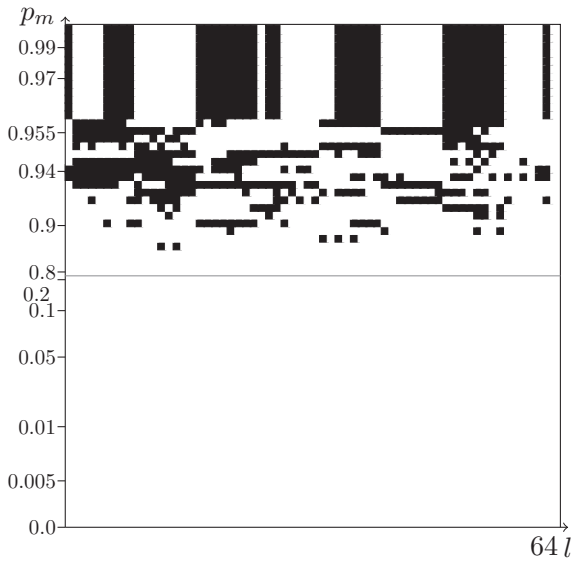


Figure A.42.: Consensus Plot: GA Royal Road Function, pop 50, nondestructive cx

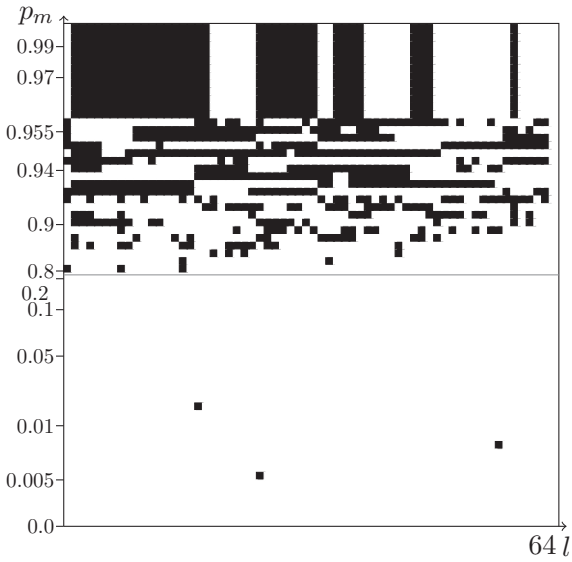


Figure A.43.: Consensus Plot: GA Royal Road Function, pop 75, destructive cx

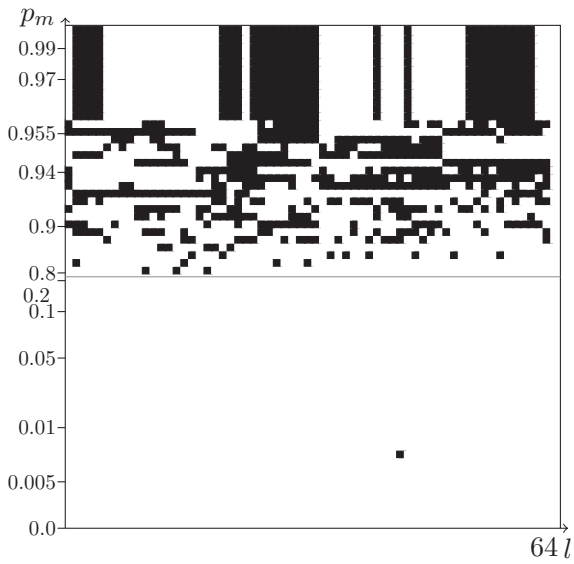


Figure A.44.: Consensus Plot: GA Royal Road Function, pop 75, nondestructive cx

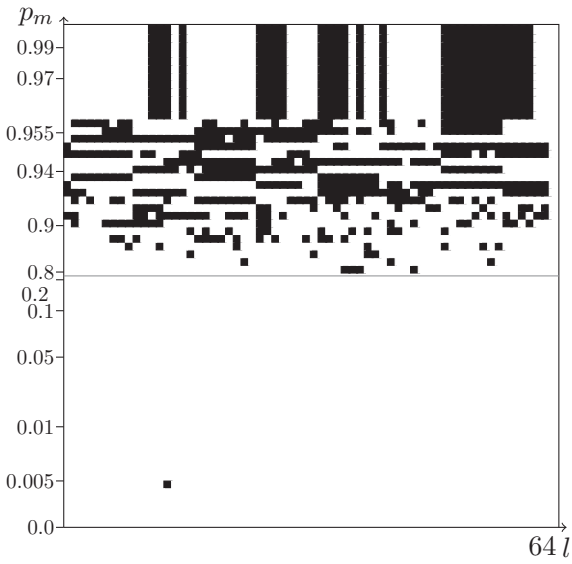


Figure A.45.: Consensus Plot: GA Royal Road Function, pop 100, destructive cx

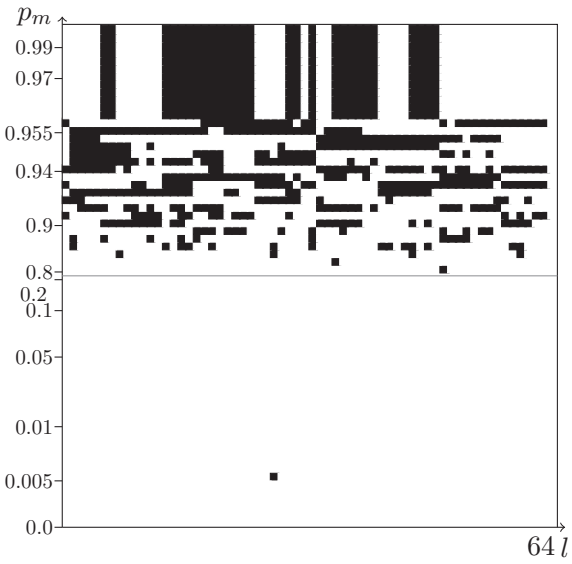


Figure A.46.: Consensus Plot: GA Royal Road Function, pop 100, nondestructive cx

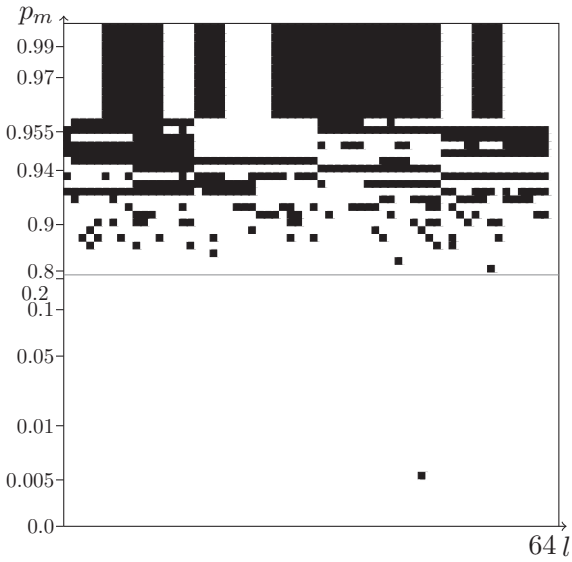


Figure A.47.: Consensus Plot: GA Royal Road Function, pop 200, destructive cx

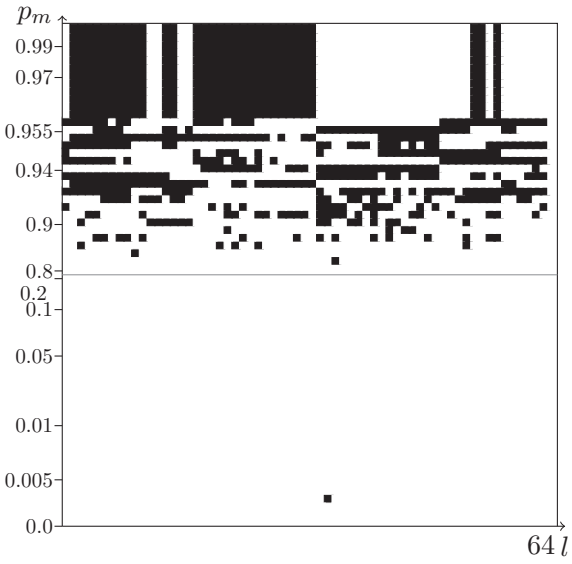


Figure A.48.: Consensus Plot: GA Royal Road Function, pop 200, nondestructive cx

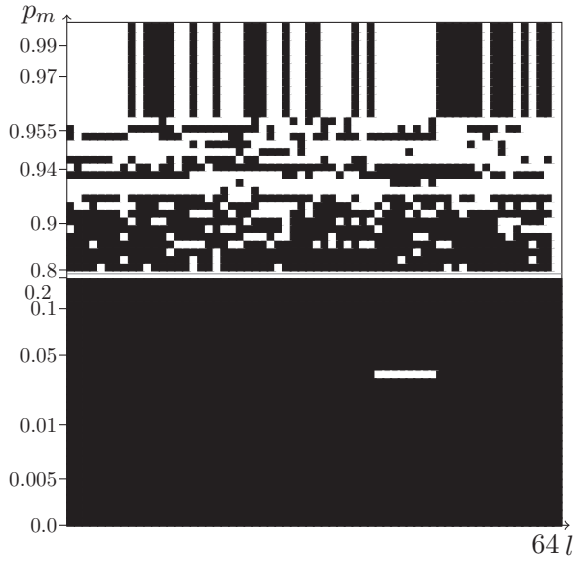


Figure A.49.: Consensus Plot: GA Trap Function, pop 10, destructive cx

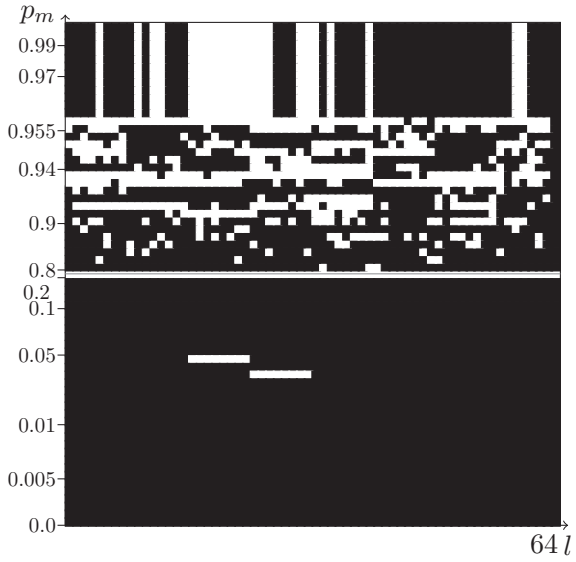


Figure A.50.: Consensus Plot: GA Trap Function, pop 10, nondestructive cx

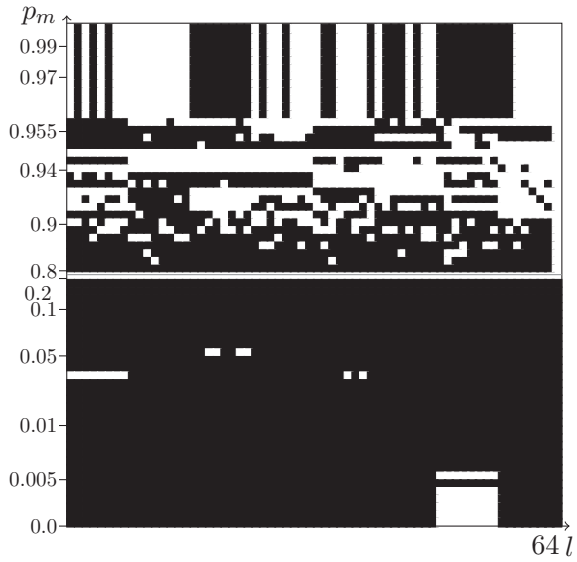


Figure A.51.: Consensus Plot: GA Trap Function, pop 25, destructive cx

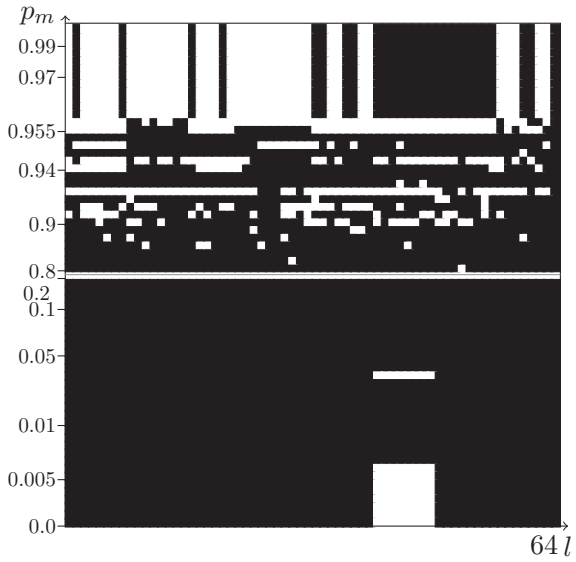


Figure A.52.: Consensus Plot: GA Trap Function, pop 25, nondestructive cx

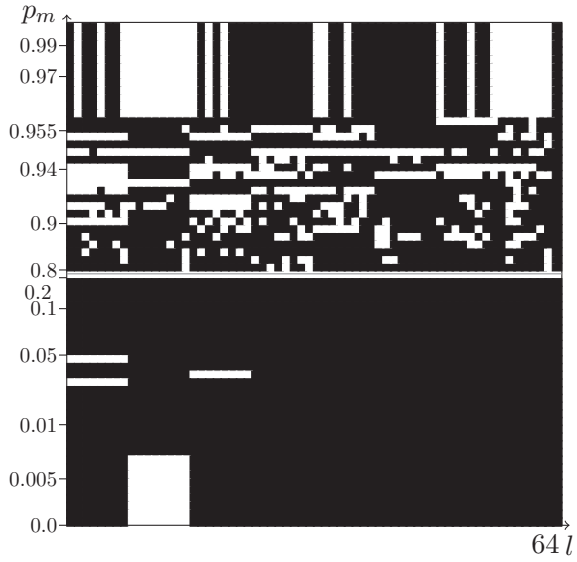


Figure A.53.: Consensus Plot: GA Trap Function, pop 50, destructive cx

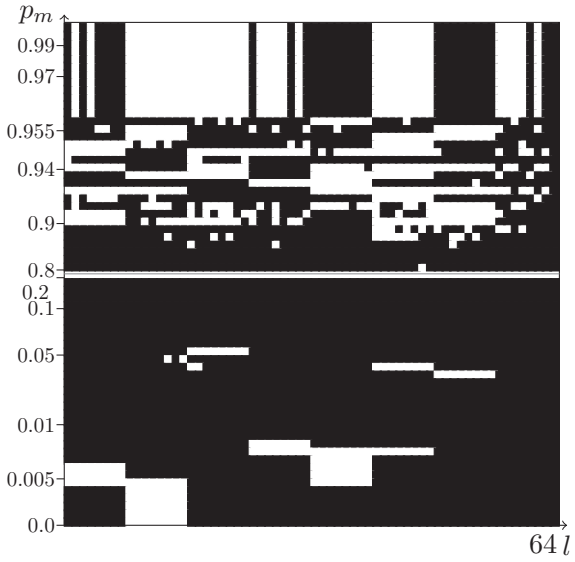


Figure A.54.: Consensus Plot: GA Trap Function, pop 50, nondestructive cx

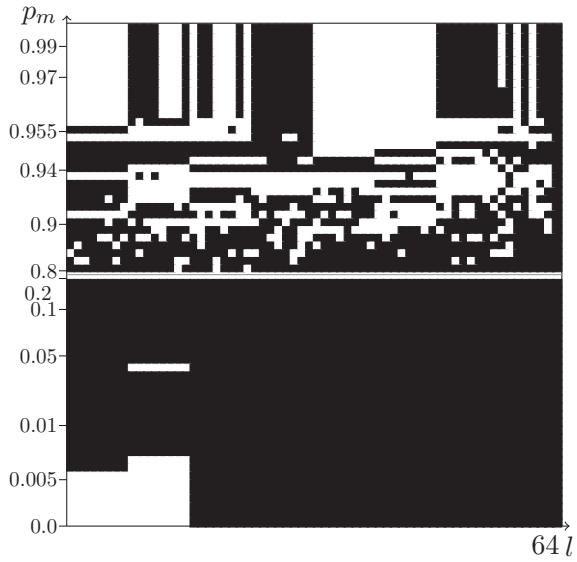


Figure A.55.: Consensus Plot: GA Trap Function, pop 75, destructive cx

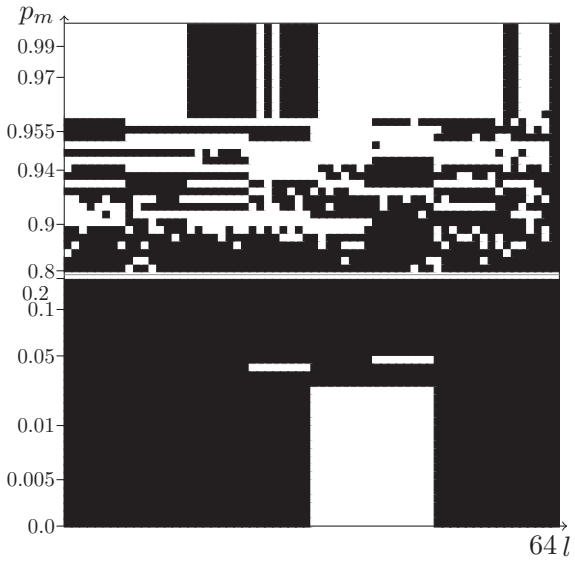


Figure A.56.: Consensus Plot: GA Trap Function, pop 75, nondestructive cx

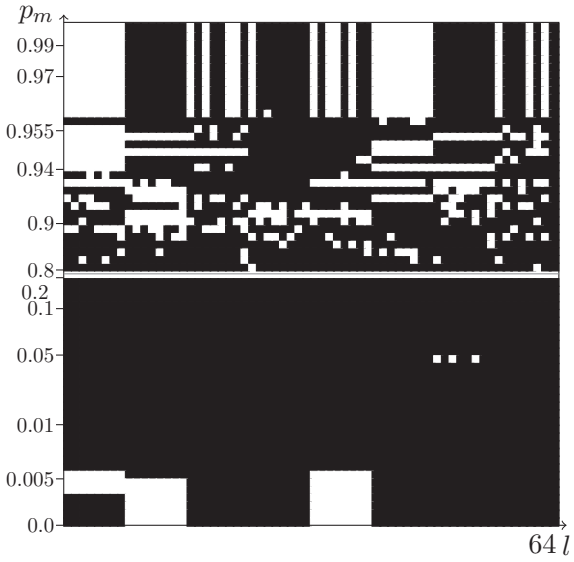


Figure A.57.: Consensus Plot: GA Trap Function, pop 100, destructive cx

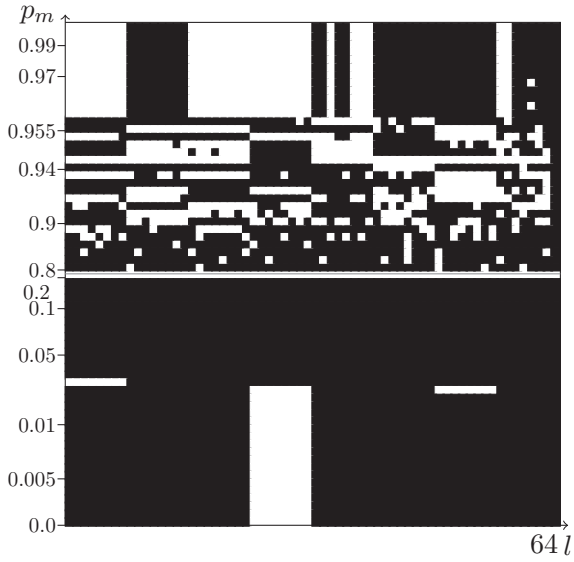


Figure A.58.: Consensus Plot: GA Trap Function, pop 100, non-destructive cx

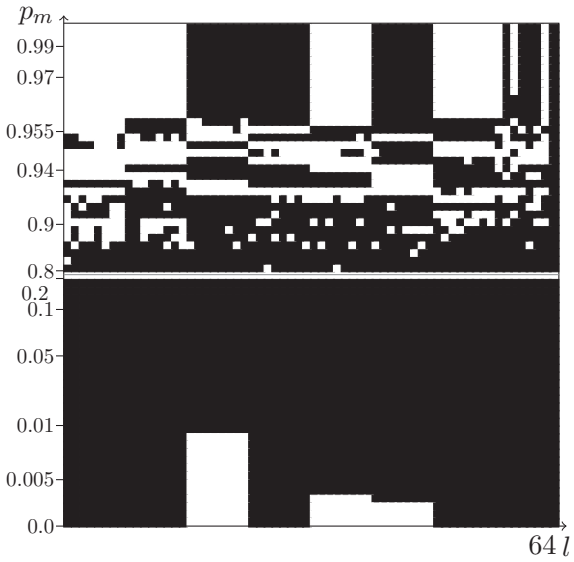


Figure A.59.: Consensus Plot: GA Trap Function, pop 200, destructive cx

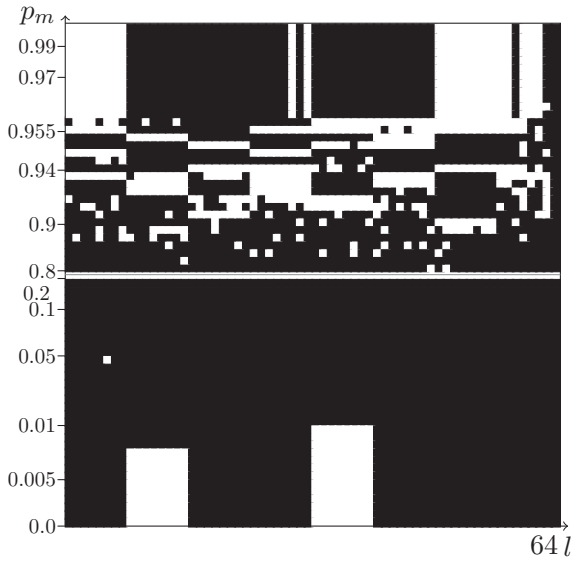


Figure A.60.: Consensus Plot: GA Trap Function, pop 200, non-destructive cx

# Bibliography


[1] Boost c++ libraries, random library, www.boost.org.

[2] Flickr, a yahoo company, www.flickr.com/services/api.

[3] H. Abdi. Holm's sequential Bonferroni procedure. *Encyclopedia of research design*, 1, 2010. Sage Thousand Oaks, CA.

[4] S. Bonhoeffer and P. Stadler. Error thresholds on correlated fitness landscapes. *Journal of Theoretical Biology*, 164:359–372, 1993.

[5] D. L. Chester. Why two hidden layers are better than one. *Proceedings of the international joint conference on neural networks*, 1:265–268, January 1990.

[6] N. E. Croitoru. Modelarea retelelor sociale folosind algoritmi genetici. Master's thesis, "Al. I. Cuza" University of Iasi, Romania, 2010.

[7] N. E. Croitoru. High-probability mutation in basic genetic algorithms. *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE*, pages 301–305, September 2014.

[8] N. E. Croitoru. High probability mutation and error thresholds in genetic algorithms. *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE*, pages 271–276, September 2015.

[9] N. E. Croitoru. Lowering evolved artificial neural network overfitting through high-probability mutation. *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, September 2016.

[10] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[11] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 5(4):455, 1992.

[12] K. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[13] K. Deb and D. E. Goldberd. Analyzing deception in trap functions. *FOGA*, 2:98–108, July 1992.

[14] L. C. W. Dixon and G. P. Szego. *Towards Global Optimization II.* New York: North Holland, 1978.

[15] S. Dreyfus. Artificial neural networks, back propagation and the Kelley-Bryson gradient procedure. *Journal of Guidance, Control and Dynamics, 1990.*, 1990.

[16] M. Eigen. Selforganization of matter and evolution of biological macromolecules. *Naturwissenschaften*, 58(10):465–523, 1971.

[17] M. Eigen and P. Schuster. *The Hypercycle: A Principle of Natural Self-Organization.* Berlin: Springer-Verlag, 1979.

[18] D. E. Goldberg, D. and R. E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. *Proc. of the 2nd International Conference on Genetic Algorithms and Their Applications*, pages 59–68, 1987.

[19] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U. Michigan Press, 1975.

[20] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

[21] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[22] A. Kun, M. Santos, and E. Szathmary. Real ribozymes suggest a relaxed error threshold. *Nature Genetics*, 37:1008–1011, 2013.

[23] M. G. Lorenz and W. Wackernagel. Bacterial gene transfer by natural genetic transformation in the environment. *Microbiological reviews*, 58(3):563–602, 1994. Am. Soc. Microbiol.

[24] Z. Luo, H. Liu, and X. Wu. Artificial neural network computation on graphic process unit. *IEEE International Joint Conference on Neural Networks*, 1:622–626, 2005.

[25] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[26] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Berlin, Heidelberg, New York: Springer-Verlag, 1992.

[27] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

[28] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. *Proceedings of the first european conference on artificial life*, pages 245–254, 1992. Cambridge: The MIT Press.

[29] R. W. Morrison and K. A. De Jong. Triggered hypermutation revisited. *Evolutionary Computation, Proceedings of the 2000 Congress on.*, 2:1025–1032, 2000. IEEE.

[30] H. Mühlenbein, D. Schomisch, , and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.

[31] M. Nowak and P. Schuster. Error thresholds of replication in finite populations mutation frequencies and the onset of muller's ratchet. *Journal of theoretical Biology*, 137(4):375–395, 1989.

[32] G. Ochoa. Consensus sequence plots and error thresholds: tools for visualising the structure of fitness landscapes. *Parallel Problem Solving from Nature*, VI:129–138, 2000.

[33] G. Ochoa. *Error Thresholds and Optimal Mutation Rates in Genetic Algorithms.* PhD thesis, COGS, The University of Sussex, Brighton, UK, 2001.

[34] G. Ochoa. Error thresholds in genetic algorithms. *Evolutionary Computation Journal*, 14(2):157–182, 2006. MIT Press.

[35] G. Ochoa, I. Harvey, and H. Buxton. Error thresholds and their relation to optimal mutation rates. *European Conference on Artificial Life (ECAL'99), Lecture Notes in Artificial Intelligence 1674, Springer-Verlagm*, 1999.

[36] L. A. Rastrigin. Convergence of random search method in extremal control of many-parameter system. *Automation and Remote Control*, 24(11):1337, 1964. PLENUM PUBL CORP CONSULTANTS BUREAU, 233 SPRING ST, NEW YORK, NY 10013.

[37] L. A. Rastrigin. *Systems of extremal control.* Zinatne, 1974.

[38] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, 1960.

[39] P. Schuster and J. Swetina. Stationary mutant distributions and evolutionary optimization. *Bulletin of mathematical biology*, 50(6):635–660, 1988.

[40] H.-P. Schwefel. *Numerical optimization of computer models.* Chichester: Wiley & Sons, 1981.

[41] W. Stolzenburg. Hypermutation: Evolutionary fast track? *Science News*, 137(25):391, June 1990.

[42] A. J. Surkan and J. C. Singleton. Neural networks for bond rating improved by multiple hidden layers. *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 157–162, June 1990. IEEE.

[43] B. L. Welch. The generalization of "Student's" problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1974.

[44] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[45] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[46] S. Yang. Genetic algorithms based on primal-dual chromosomes for royal road functions. 2002.