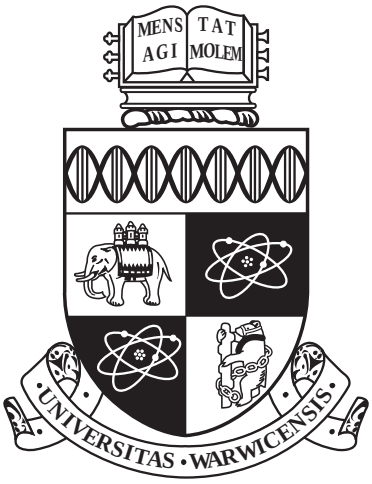


Asynchronous Parallel Numerical Optimization

by

Joy Tolia

MA469 Fourth Year Project

Submitted to The University of Warwick

Mathematics Institute

March, 2015

THE UNIVERSITY OF
WARWICK

Abstract

This work examines the process of unconstrained optimization, using ideas based on genetic algorithms such as different breeding types; mutation and crossover. We look at numerical methods which are not gradient based to allow for highly non smooth continuous function optimization. We will assume, as is often the case in scientific computing, that the function evaluation takes a lot of time and therefore we want to parallelize that part of the algorithm.

This uses the master-worker paradigm where the workers run in parallel and their only job is function evaluation. The master does everything else such as population generation. We make the algorithm asynchronous by starting a new generation before gaining all the information from the previous generation. Then we implement this algorithm in MATLAB using SPMD and apply it to a set of test functions to study various parameters.

We conclude for the functions we have tested, that the asynchronous model works well with the breeding type mutation but not so well with crossover. We then examine a higher dimensional function where we learn more about the decay of jump sizes in mutation. Finally we explore using both breeding types, crossover and mutation, together.

Contents

Acknowledgements

I would like to express my deepest gratitude to my project supervisor Tim Sullivan for helping and guiding me throughout this project. I am grateful for all the support from my fellow undergraduates, especially Rosie Ferguson, Calvin Khor, Tom Reddington and Jack Betteridge. I would like to extend my appreciation to my family for their help and encouragement.

1 Introduction

1.1 Optimization

Let us first start with optimization, we look at finding the minima and minimizers of functions. Let $f : D \to \mathbb{R}$ be a function where $D \subset \mathbb{R}^d$ for some $d \in \mathbb{N}$ with at least one global minimum. We will make an assumption that $D \subset \mathbb{R}^d$ is compact. The problem is to find a point in the domain that minimizes the function, we want to find $x^\star \in D$ so that:

$$x^\star = \underset{x \in D}{\operatorname{argmin}} f(x)$$

There is also constrained optimization, where we are doing the above and the domain is also restricted (possibly in multiple ways) with the following constraints:

$$\begin{aligned} g_i(x) &= c_i \quad \text{for } i = 1, \ldots, n \\ h_j(x) &\geq d_j \quad \text{for } i = 1, \ldots, m \end{aligned}$$

where $g_i, h_j : \mathbb{R}^d \to \mathbb{R}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$ for $n, m \in \mathbb{N}$.

In this project we will only be looking at unconstrained optimization. We want to focus on using numerical methods to find minimums of functions. One of the ways we can do this is by using ideas from a certain class of algorithms called genetic algorithms. This class of algorithms is very useful as gradient based methods tend to fail in the case of highly non-smooth functions.

1.2 Genetic Algorithm

Genetic algorithms are based on the process of evolution, the structure is explained in [1]. We start with a *population* which is a set of *elements* then we make a new *generation* by breeding the *fittest* elements. Each increase in generation should help us get a fitter population. The fitness of an element is calculated using an *objective function*. In our case we will have a random set of points in D as our initial population. Then we use an optimization algorithm on those points to get a new population. For example choosing the best two points who give the lowest value of the function and finding random points around these to make a new population. Our objective function will be the minimizing function. We will use the following notation, where we are trying to optimize the function f:

$$f : D \to \mathbb{R}, \quad D \in \mathbb{R}^d$$

$$X_i = \begin{pmatrix} \vdots & & \vdots \\ x_1^{(i)} & \dots & x_p^{(i)} \\ \vdots & & \vdots \end{pmatrix} \in \mathbb{R}^{d \times p}$$

Where p is the size of the population, $x_1^{(i)}, \dots, x_p^{(i)} \in D$ and X_i is the population for the i-th iteration. $P : \mathbb{R}^{d \times p} \times \mathbb{R}^{1 \times p} \to \mathbb{R}^{d \times p}$ is the population generating function which will use some kind of breeding process to make a new population of elements given the previous generation. Algorithm 1.1 shows how we can use genetic algorithms for optimization.

Algorithm 1.1 Optimization Genetic Algorithm

Input: X_0

Output: X

Initialise $X = (x_1, \dots, x_p) = X_0 \in \mathbb{R}^{d \times p}$
Initialise $E = (e_1, \dots, e_p) \in \mathbb{R}^{1 \times p}$
while Termination not reached **do**
  for $j = 1 \leq p$ **do**
    $e_j = f(x_j)$
  end for
  Check for termination
  $X = P(X, E)$
end while

Figure 1 is the flow chart for Algorithm 1.1.

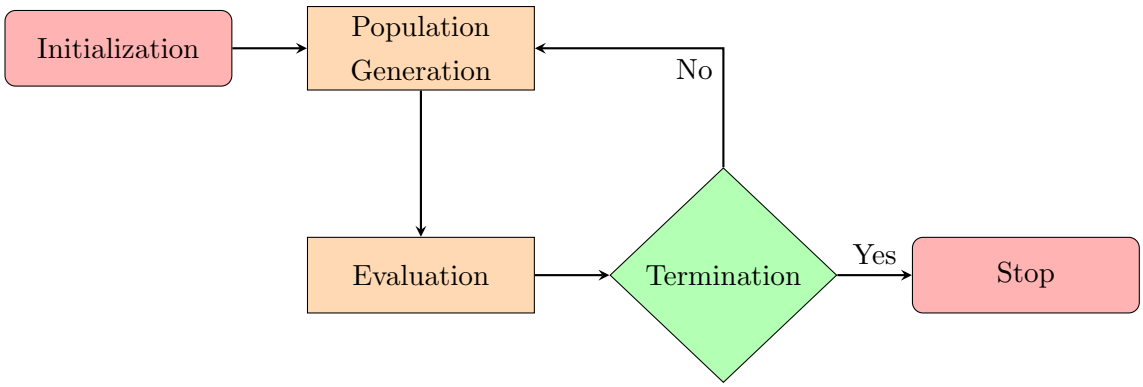


Figure 1: Flow chart for optimization genetic algorithm

We will assume, as is often the case in scientific computing, that the evaluation part of the algorithm takes a lot more time than population generation and constraints. So we want to try and work around that. Suppose we can run the evaluation in parallel then we could speed up our algorithm, for example if the evaluation is an expensive PDE solver then parallelising would give us a significant speed up. As displayed in Figure 1, we have

to wait for the whole population to be evaluated before we can start a new generation. This is obviously very costly in terms of time. To work around this, we can start a new generation once a certain number of elements have been evaluated. This would give us less information from each generation as fewer elements are evaluated. However, we reach a higher number of generations which could mean that we find the minimum faster. We would like to study whether doing this makes our algorithm faster than an algorithm where a full population is evaluated before starting the next generation.

1.3 Asynchronous Parallel Algorithm

Let us first look at the algorithm we are working with. We split the cores of our computer into one master and the rest workers, explained further in [2]. The master is in charge of generating the population, sending and receiving messages to and from the workers and storing work. The sole purpose of the workers is to do the work sent by the master. The algorithm is based on communication to and from the workers and the master. The workers never have to communicate with each other. In our case the work that the workers will be doing is evaluating a function for a given element.

This is better than the alternative of not having a master and using a peer to peer system where information is communicated between all workers. Because it reduces the size of communications by keeping all the information in one core (master) and sending as little information as possible to the other cores (workers). For us the information would be the population from the different generations. This also avoids two workers doing the same work in the case where relevant information was not communicated in time. The downside is that the master could be idle a lot of the time whilst waiting for the workers to send back work.

We will first look at a very general asynchronous parallel algorithm not necessarily for optimization. Algorithm 1.2 is the algorithm for the master.

Algorithm 1.2 Asynchronous Parallel Master Code

```
Generate initial population
while Termination is not reached do
    Receive message from any worker
    if Worker requested work then
        if Check there is work to give out then
            Send work to that worker
        else
            Send stop message to the worker
        end if
    else if Worker sent finished work then
        Store the finished work
        Check termination condition
        if Termination reached then
            Send stop message to the worker
        end if
        if Enough work has been received to generate the next then
            Generate a new generation of population
        end if
    else if Worker has received stop message then
        Once all workers send this message, termination is reached
    end if
end while
```

The different parameters we can work with here are the number of workers, how to pick an element for evaluation and how much work needs to be done before we start a new population. Algorithm 1.3 is the algorithm for the worker.

Algorithm 1.3 Asynchronous Parallel Worker Code

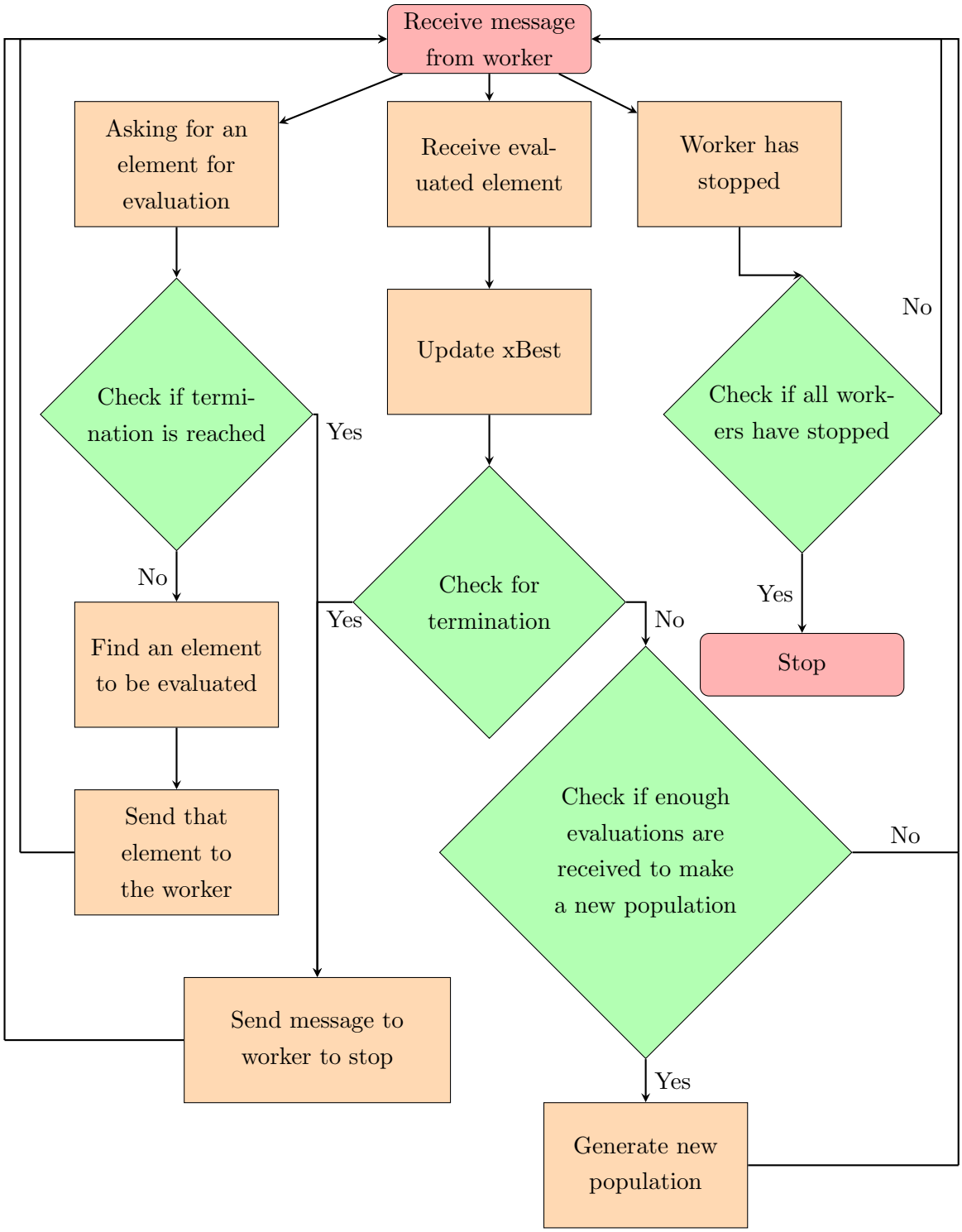
```
while Stop message is not reached do
    Send message to master requesting work
    Receive message from master
    if Work is received then
        Do work
        Send finished work back to the master
    else if Stop message is received then
        Termination for worker is reached
    end if
end while
```

This is parallel is because the workers can all do work at the same time. The sending and receiving of messages is not parallel however once the workers have received their work, they can all be working in parallel. The asynchronicity comes from the fact that we don't wait for all the work to be received back from the current population before making the next generation and work is started on that generation as soon as it is made.

2 Algorithms

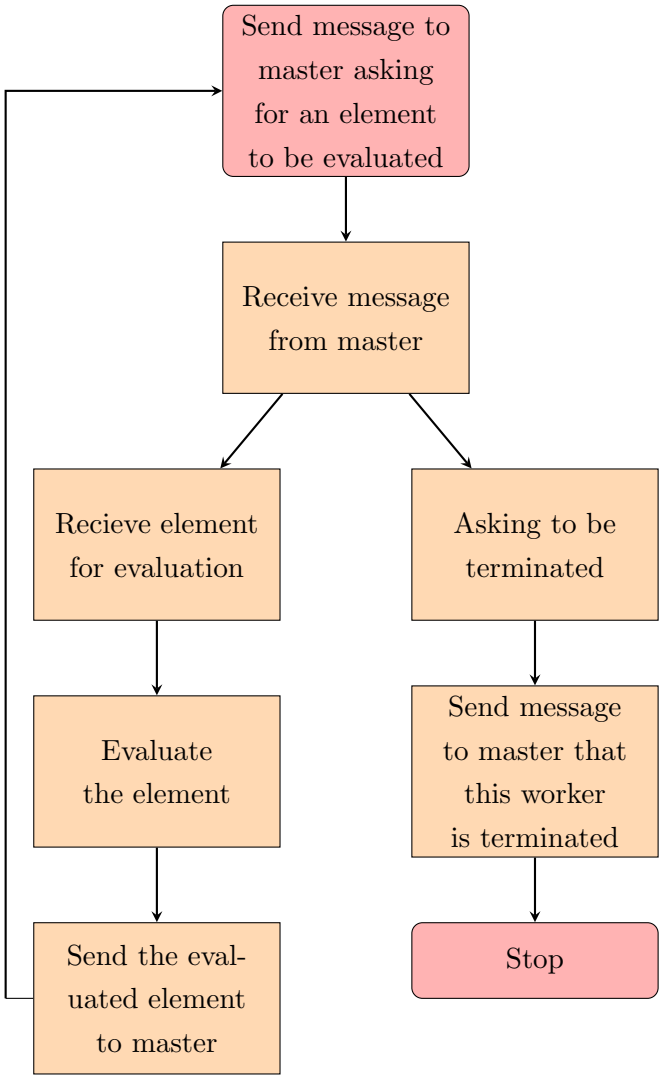
Let us start by looking at the algorithm for asynchronous parallel numerical optimization. We are now in the master-worker paradigm. The following is the flow chart for the master:

Figure 2: Algorithm for master for optimization



The following is the flowchart for the worker:

Figure 3: Algorithm for worker for optimization



xBest is what stores the fittest elements (which for us are the elements with the lowest evaluated values) throughout the algorithm. From Figure 2, some of the potential parameters to analyse are:

- Number of workers.
- Number of elements in xBest, we do this as a ratio of the population size.
- Number of elements we need to evaluate before we can start the next generation, again we do this as a ratio of the population size.
- The process of finding an element to be evaluated, this is where we introduce how much priority is given to the latest generation.

We also have some important parts of the algorithms to discuss which are:

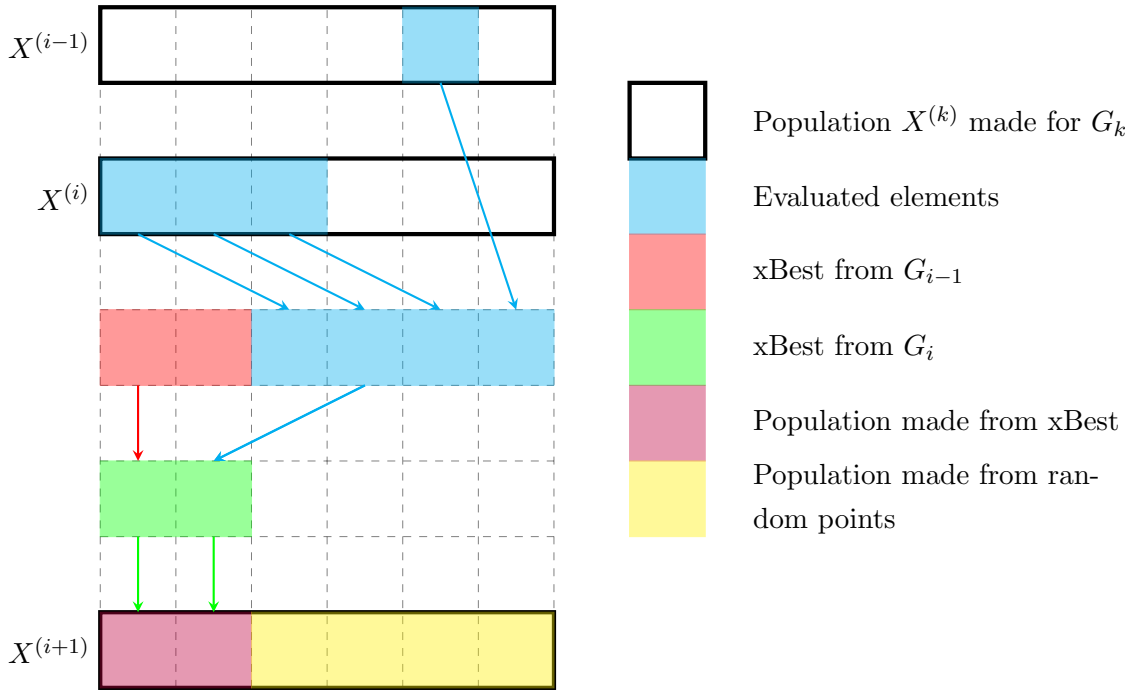
- How doe we generate the new population?
- What are the termination criteria?

2.1 Generating Population

We use xBest to generate part of our population and the rest we pick uniformly random points in the domain. We will use xBest to keep track of the fittest elements. We update the xBest when elements are evaluated and when enough elements have been evaluated, we use the updated xBest to generate part of our population. In this project, the number of elements in the population made from the xBest will exactly be the number of elements in xBest i.e. the size of xBest.

Firstly, let us look at the bigger picture and study how we generate a new population and then we will analyse the details about how we breed elements from xBest. Figure 4 shows how we use xBest together with the evaluated elements to make a new xBest from which we then make a new population.

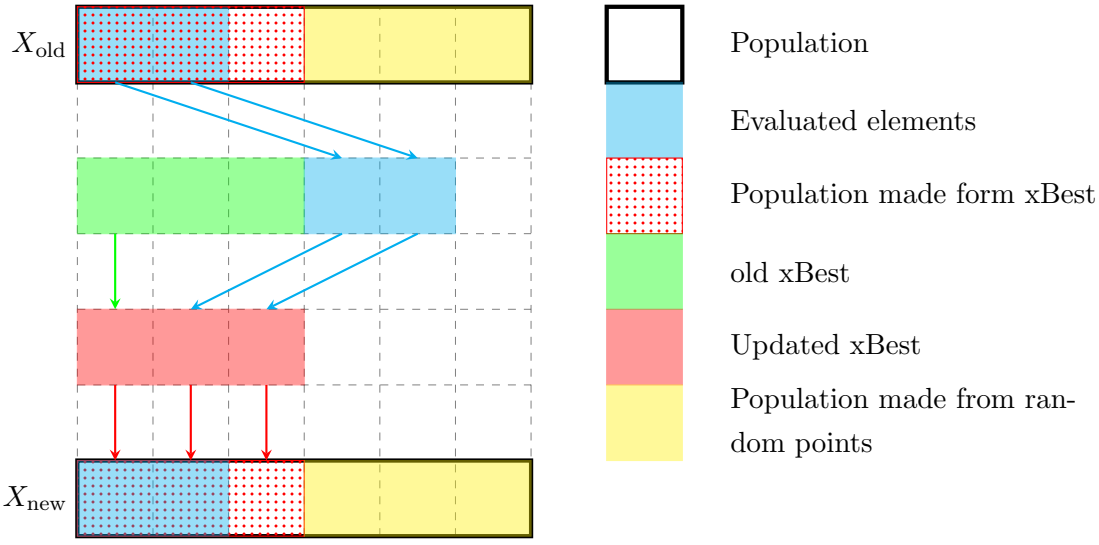
Figure 4: Generating a new population; G_k represents $k-$th generation



Once the population is made, it needs to be randomly permuted, this is because when we choose an element for evaluation, we choose the first element in the population that has not been evaluated or is not currently being evaluated. When the population is made, the first part of the population is made from the xBest and the rest are random points

in the domain. This means that if the population is not permuted randomly and if the number of elements needing to start the next generation is smaller than the size of the xBest then the algorithm will only ever evaluate the elements of the population which are made from the xBest. This would mean we could get stuck in local minimas easily or as the initial xBest is made from the initial population, we might have our xBest very far from the global minimum hence we might never reach it as we are never evaluating the random elements in our population and the elements made from xBest are very close to xBest. This can be seen more clearly in Figure 5.

Figure 5: Explanation of the problem when not randomising the population



2.1.1 Breeding

There are two ways we can breed from xBest; *mutation* and *crossover*. Mutation is where we take a random *jump* from each element in xBest to get a new element. Crossover is where we take two or more parent elements from xBest and make a child element with a method that encodes information about the parent elements.

The jump function gives us a random element close to the element we started with. We will use the method in Algorithm 1.4.

Algorithm 1.4 jump (Making a jump for our element)

Input: x (one element in xBest), genNumber (the generation iteration we are in), bounds (of our domain)

Output: y

```
d = dimension of x
for i = 1 : d do
    Sample r from a standard normal distribution
    r = r×temperature(genNumber)
    y_i = x_i + r
    if y_i outside bounds then
        Set y as the respective bound
    end if
end for
```

We want to assume that as the number of generations increases, we are getting closer to the minimum and hence we want to take smaller jumps to get closer to the true minimum. The temperature function is used to make smaller jumps as the number of generations increases. We use the following simple temperature function:

$$\text{temperature}\,(n) = \frac{1}{10 \times n}$$

The other method of breeding we shall discuss, is crossover. The method we will use is fitting a Gaussian distribution to the elements in xBest and then sampling from this distribution. This is a crossover method because we are encoding information from all the parent elements to produce the child element. We do this by calculating the mean and the standard deviation of the elements in xBest which we then use as the mean and the standard deviation of the Gaussian distribution to sample from.

An issue that needs to be considered is when we have multiple global minima which are far apart as this will give a Gaussian distribution with the mean not close to any of the minima, making it difficult to converge to one of the minima. There are some methods to help these types of situations. One of them is using a Gaussian mixture model to sample from. This is when we have multiple Gaussian distributions added up together and rescaled to form the Gaussian mixture model. However it is difficult to calculate the number of components of the Gaussian mixture model from a given data set, more information on some of these methods is given in [3].

2.2 Fading Memory or Priority to the Latest Generation

We want to see if giving priority to the latest generation's population for element evaluation makes any changes to our convergence to the minimum. As our algorithm is asynchronous, we can pick an element from any generation for evaluation, as long as they have not all been evaluated. We would like to build a probability measure that we can sample from.

We do this using the geometric distribution which we then truncate and rescale to form a distribution on a finite state space. Given $p \in [0,1]$, this is defined by a discrete probability measure $\mu : \mathbb{N}_0 := \{0, 1, 2, \dots\} \to [0,1]$ by:

$$\mu(n) = p(1-p)^n, \quad n \in \mathbb{N}_0$$

Suppose the latest generation is $N \in \mathbb{N}_0$. Then we have:

$$\begin{aligned} \mu\left(\bigcup_{i=0}^{N}\{i\}\right) &= \sum_{i=0}^{N}\mu(i) \\ &= \sum_{i=0}^{N} p(1-p)^i \\ &= \frac{p\left[1-(1-p)^{N+1}\right]}{1-(1-p)} \\ &= 1-(1-p)^{N+1} \end{aligned}$$

Let us define $\mathbb{P} : \{0, 1, \dots, N\} \to [0,1]$ by:

$$\mathbb{P}[n] = \frac{\mu(N-n)}{1-(1-p)^{N+1}} = \frac{p(1-p)^{N-n}}{1-(1-p)^{N+1}}, \quad n \in \{0, 1, \dots, N\}$$

This is a discrete probability measure because $\mathbb{P}[\{0, 1, \dots, N\}] = 1$ by construction and the additivity of $\mathbb{P}$ holds by the additivity of μ. Sampling from the distribution generated by the measure $\mathbb{P}$ is exactly what we want to use to choose what generation we want to evaluate from, given the priority probability p. We can see how this in implemented in Section 5.1.5. When $p = 1$ we always pick an element from the latest generation for evaluation. Figures 6-9 show histograms of sampling from $\mathbb{P}$ with different priority probabilities.

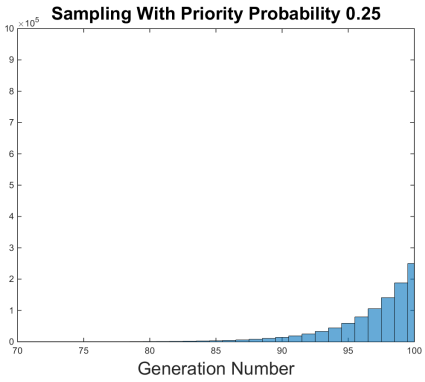


Figure 6: 1×10^6 samples from the measure $\mathbb{P}$ with priority probability of 0.25 and latest generation 100

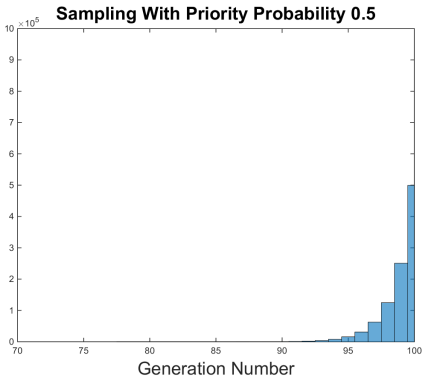


Figure 7: 1×10^6 samples from the measure $\mathbb{P}$ with priority probability of 0.5 and latest generation 100

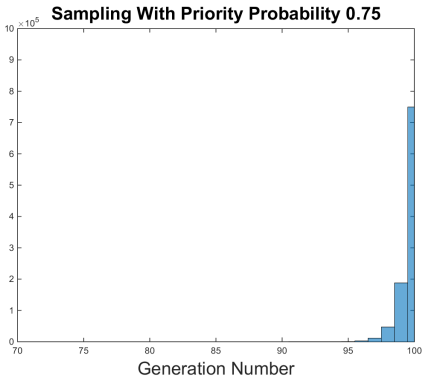


Figure 8: 1×10^6 samples from the measure $\mathbb{P}$ with priority probability of 0.75 and latest generation 100

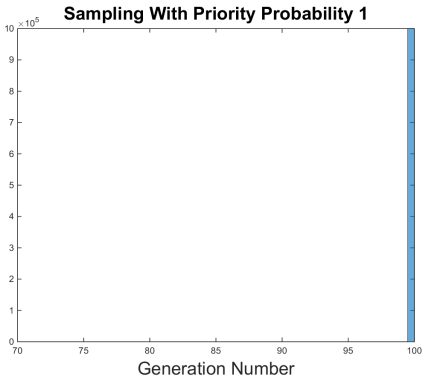


Figure 9: 1×10^6 samples from the measure $\mathbb{P}$ with priority probability of 1 and latest generation 100

2.3 Termination Criteria

In finding the minimizers, we find the minima and vice versa. So without loss of generality, we will find the minima in our implementation. There are multiple ways of checking for termination. Here are some of the following:

1. Maximum number of generations
2. Maximum amount of time
3. Tolerance on absolute changes of successive elements of xBest
4. Tolerance on the absolute difference of evaluations of elements in xBest against a given minimum (here we assume we already know the minimum)

As we will be testing our algorithms with test functions, we will know the minima apriori so the termination criteria will be working with are (2) and (4) from the above list.

2.4 Implementation

This algorithm was implemented in MATLAB using the SPMD statement which lets you run in parallel and has communication between the parallel workers in a similar way to MPI in C. The code for this can be seen in Section 5.1.

3 Testing

3.1 Number of Workers

We will look at how long it takes to evaluate a function a certain number of times with a given number of workers so we will not be looking at convergence results yet. Table 1 shows the different parameter we use to get the data.

Table 1: Parameters used to get the data

Distribution for the time it takes to evaluate the function	max(N(1,1),0)
Number of workers	1-10
Repeats	5
Population size	50
Number of generations	100
Total number of evaluations	5000

This is the data we are getting:

Table 2: Times for 5000 Evaluations

Number of Workers	1	2	3	4	5
Time ($\times 10^3$ seconds)	5.2019	2.7354	1.8357	1.3898	1.1107

Number of Workers	6	7	8	9	10
Time ($\times 10^3$ seconds)	0.9216	0.7907	0.6922	0.6169	0.5556

Suppose we denote $T(w)$ for $w \in \mathbb{N}$ as the time taken to evaluate the function 5000 times by w number of workers.

Definition 1.1. We define the *speed up* S for $w \in \mathbb{N}$ workers by:

$$S(w) = \frac{T(1)}{T(w)}$$

Definition 1.2. We define the *efficiency* E for $w \in \mathbb{N}$ workers by:

$$E(w) = \frac{S(w)}{w} = \frac{T(1)}{w \times T(w)}$$

Remark 1.3. As the name suggests, speed up tells us how much faster or slower our algorithm runs given the number of workers. Efficiency can be thought of as the speed up per worker.

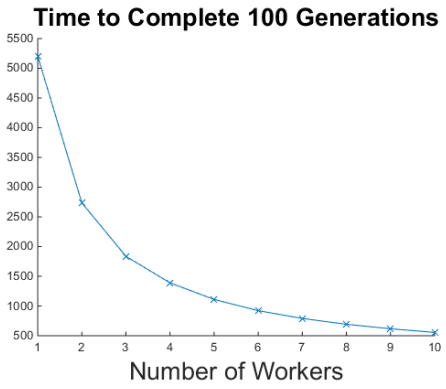


Figure 10: Time take to evaluate the function 5000 times

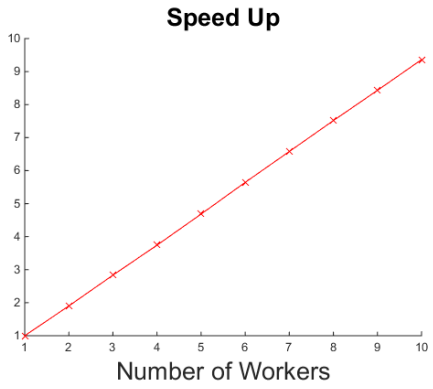


Figure 11: Speed up from increasing number of workers

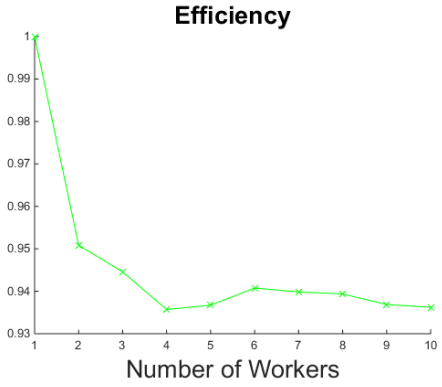


Figure 12: Efficiency from increasing number of workers

3.1.1 Analysis

- We can see that the time for evaluation decreases as the number of workers increase which means our parallelism is working well.
- We find that the speed up is linear which means our algorithm speeds up by the number of numbers which is very good for us. This means that communication has hardly any effect on the time it takes for our algorithm to run.
- In terms of efficiency, we can see that it is decreasing at first and then is nearly constant after. This means our speed up per worker is decreasing at first and then is constant, this is another indicator that the communication in our algorithm is not becoming an issue when increasing the number of workers (up to 10). This might be because to evaluate the function it takes approximately 1 second which will be much bigger than any of the communication or serial part of our algorithm i.e. the master's algorithm.

- All of this means our parallelism is good and helpful in reducing the time to run our algorithm.

3.2 Description of the Testing

For the rest of this section, we will adjust different parameters that we have and analyse the results we get. We will be working with a set of test functions $f : [-5, 5]^2 \in \mathbb{R}^2 \to \mathbb{R}$, until the last subsection, which can be seen in the appendix.

We will be working with 8 different test functions, all which which can be seen in Section 5.2. In Section 5.2.9, we can see different sample paths obtained from mutation breeding. They give us an idea of how the algorithm moves around the domain searching for minimizer. Table 3 explains why we use these functions.

Table 3: Description of the test functions

Function	Description
1	This is a very thorny function with lots of local minima to check if the algorithm gets stuck in the local minima.
2	This is a smooth function which should be easy to optimize.
3	This is a shallow function and good for checking if the algorithm gets stuck.
4	Similar to function 3 but has a different shape of the base.
5	This is a wavy function again with multiple local minima.
6	This is a mostly flat function with a small part which drops to a minimum.
7	This function has multiple global minima near the boundary of the domain and also has some local minima.
8	This is a multi dimensional function where we can define the number of dimension of the domain, it has a lot of local minima.

For the rest of this section, we will be adjusting different parameters and studying the following properties:

1. Time taken for the algorithm to converge to a minimum or capped with a given time. This will tell us if our algorithm is faster or slower when adjusting a parameter. We should note, we that we average over all repeats, even if the repeat has not converged to the minimum. In doing this, we implicitly penalise the repeat if it has not converged as it will have reached the maximum time allowed for the algorithm to run.

2. Number of generations reached by the algorithm. This will show us how many generations were needed to get to for convergence to a minimum, if the algorithm converged to a minimum. For different parameters, number of generations will have similar or different relation with time taken.

3. Percentage of the repeats that the algorithm converged to a minimum. This will tell us how often the algorithm is converging to the minimum for different parameters.

4. Error plots for different levels of a parameter for each function. Fixing the parameter to a certain level and fixing the the function, we look at all the repeats that have converged then look at the error of the repeat with the middle value in terms of the number of generations reached and plot that. If there are no repeats that have converged then nothing will be plotted. Here we can compare different levels of a parameter by looking at the *middle* error decay of the algorithm for the given function.

5. Error spread plots different levels of a parameter for each function. Again fixing the parameter to a certain level and the function, we look at all the repeats that have converged then look at maximum and minimum error of the repeat for each generation and shade the region between. If there are no repeats that converged then nothing will be plotted. If only one repeat converged then a line is plotted instead of a region. From this we can see how much the error decay varies, when lower, the information from the error decay is more accurate.

6. We can combine all the above properties to get information on how much different levels of a parameter's error decay overlap each other and from that we can pick a level out of two which have similar error decays but one might converge to the minimum faster than another on average or has a higher percentage of convergence to a minimum.

3.3 Fading Memory or Priority to the Latest Generation - Mutation

Table 4: Parameters used to get the data

Ratio of xBest to the population size	0.5
Ratio of number of elements to evaluate before starting a new generation to the population size	0.7
Number of workers	5
Repeats	20
Population size	50
Priority to the latest generation	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Maximum Time	60 seconds
Tolerance	1×10^{-6}
Functions used	1, 2, 3, 4, 5, 6, 7
Breeding type	Mutation

Figure 13: Results for adjusting the priority given the latest generation with mutation

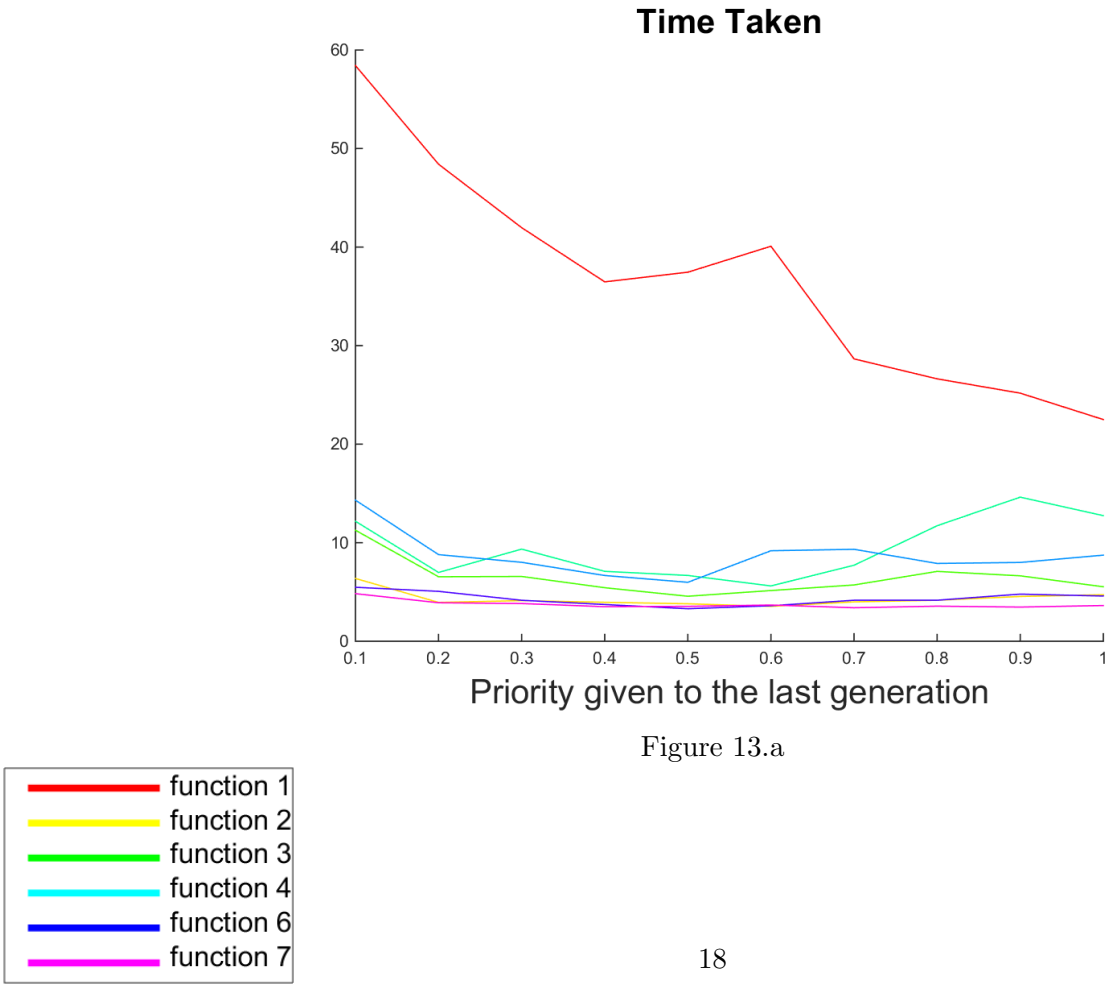


Figure 13.a

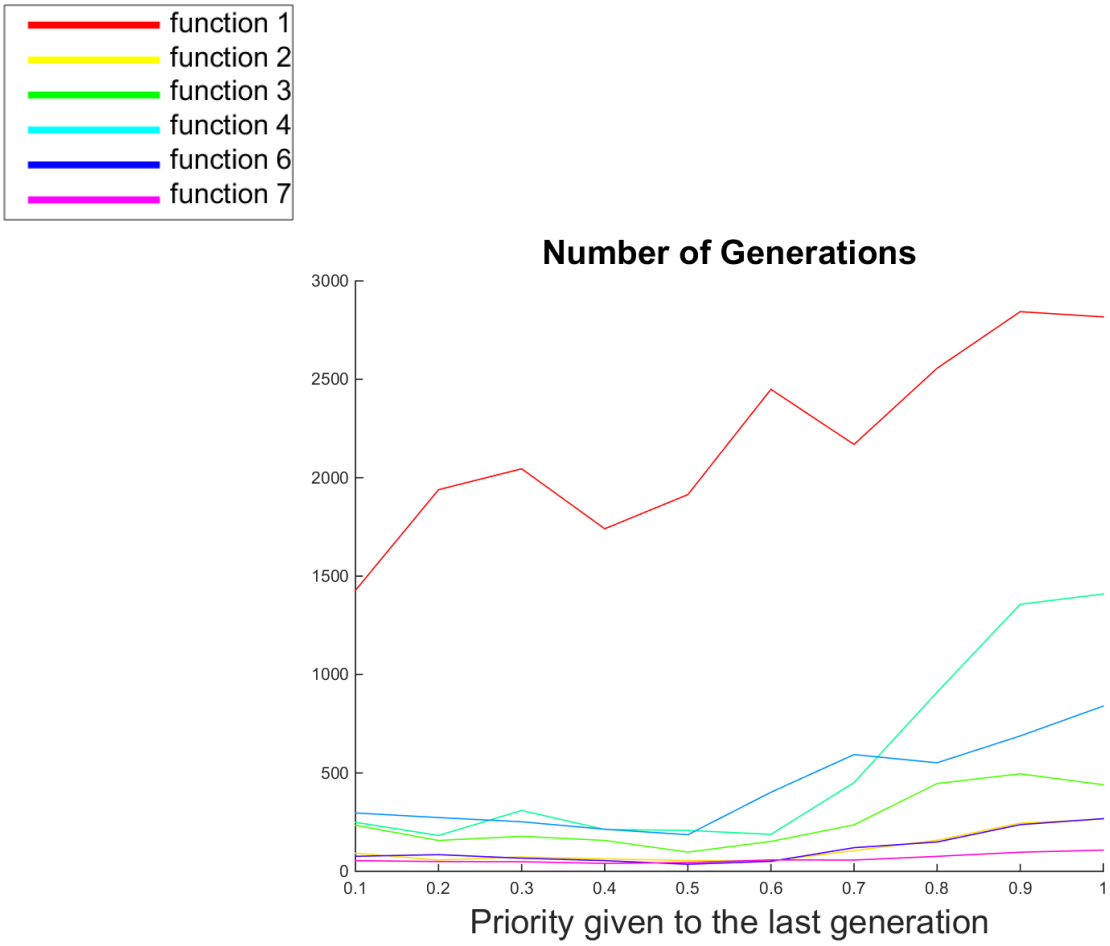


Figure 13.b

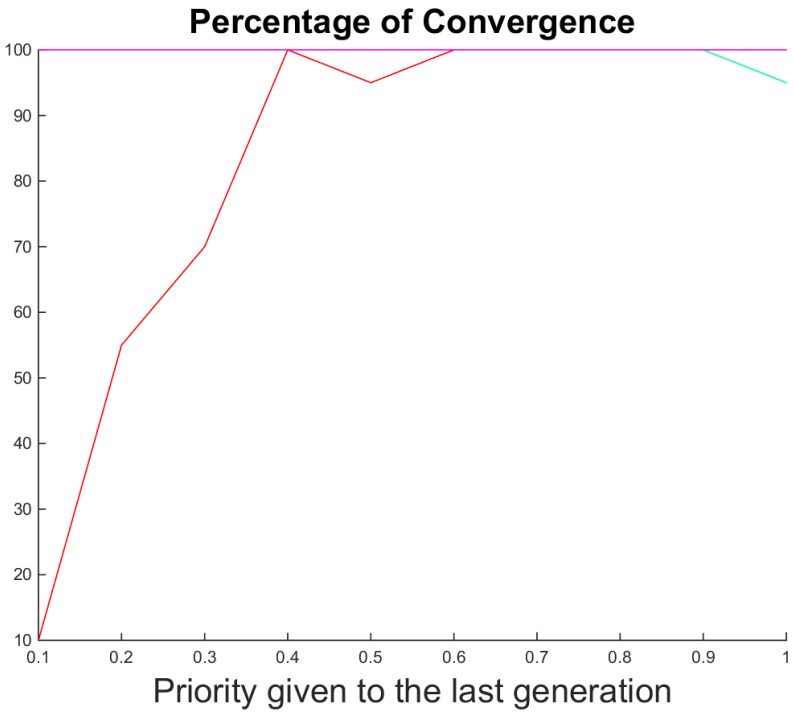


Figure 13.c

- From Figure 13.a, we can see that the time taken for the algorithm for function 1 reduces with priority to te latest generation however for other functions, it is mostly flat but increases past priority level of 0.7. We also have to remember that the function here took very little time to evaluate but if we are working with functions that take a lot of time to evaluate, the larger the ratio the longer it would take run one generation, this means that between 0.4 and 0.6, the priority gives the best results.
- Figure 13.c shows that all functions except function 1 converge nearly every time. Function 1 has lower percentage of convergence from lower priority because it needs information from the latest generation to jump out of local minimas.
- We can see, for most functions other than function 1, that the different priority levels do not make too much of a difference in terms of time taken and percentage of convergence.
- In conclusion, priority levels between 0.4 and 0.6 seem to give the best results for most functions. However, for functions like function 1, it seems that the higher the priority level, the better results are obtained.

Figure 14: Errors for adjusting the priority given the latest generation with mutation

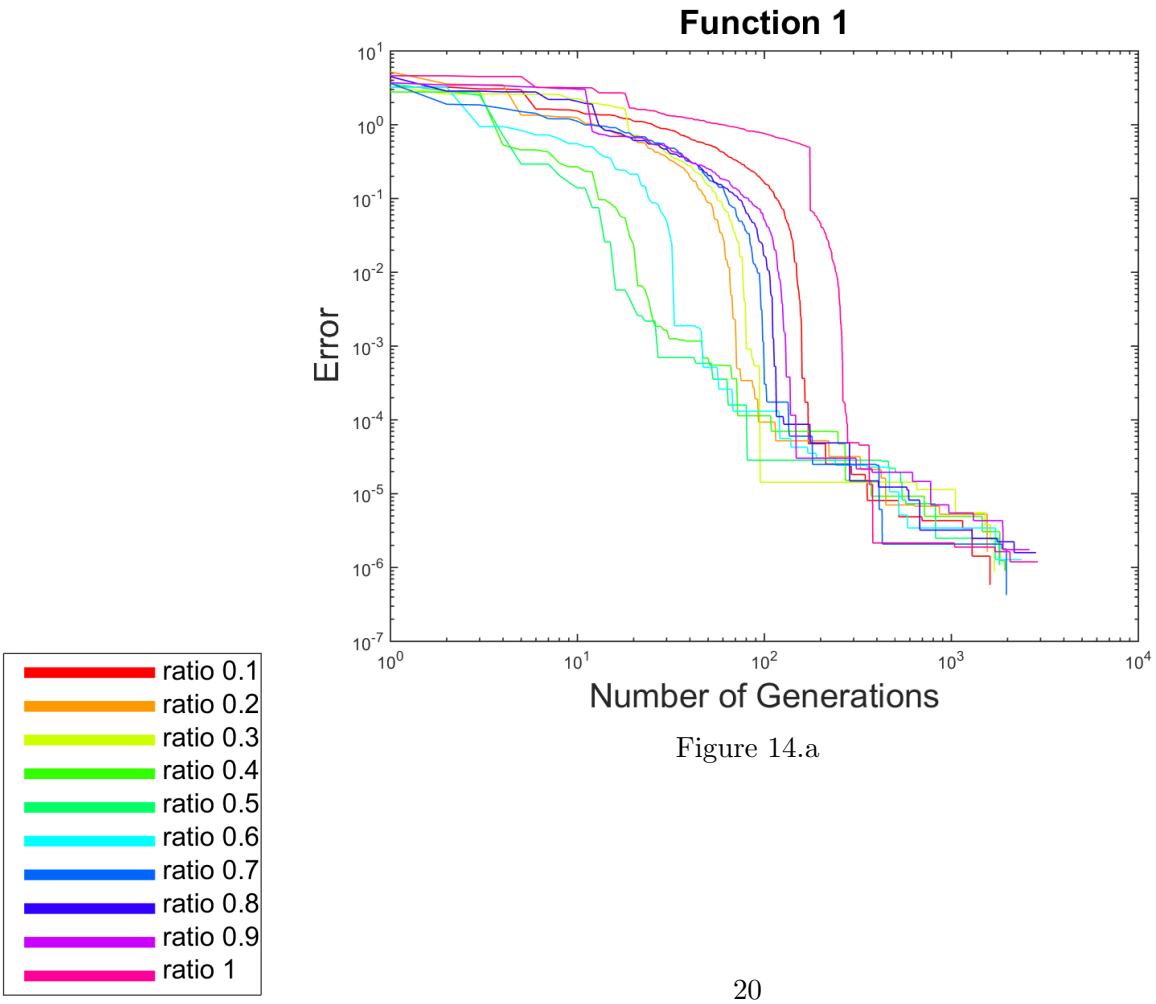


Figure 14.a

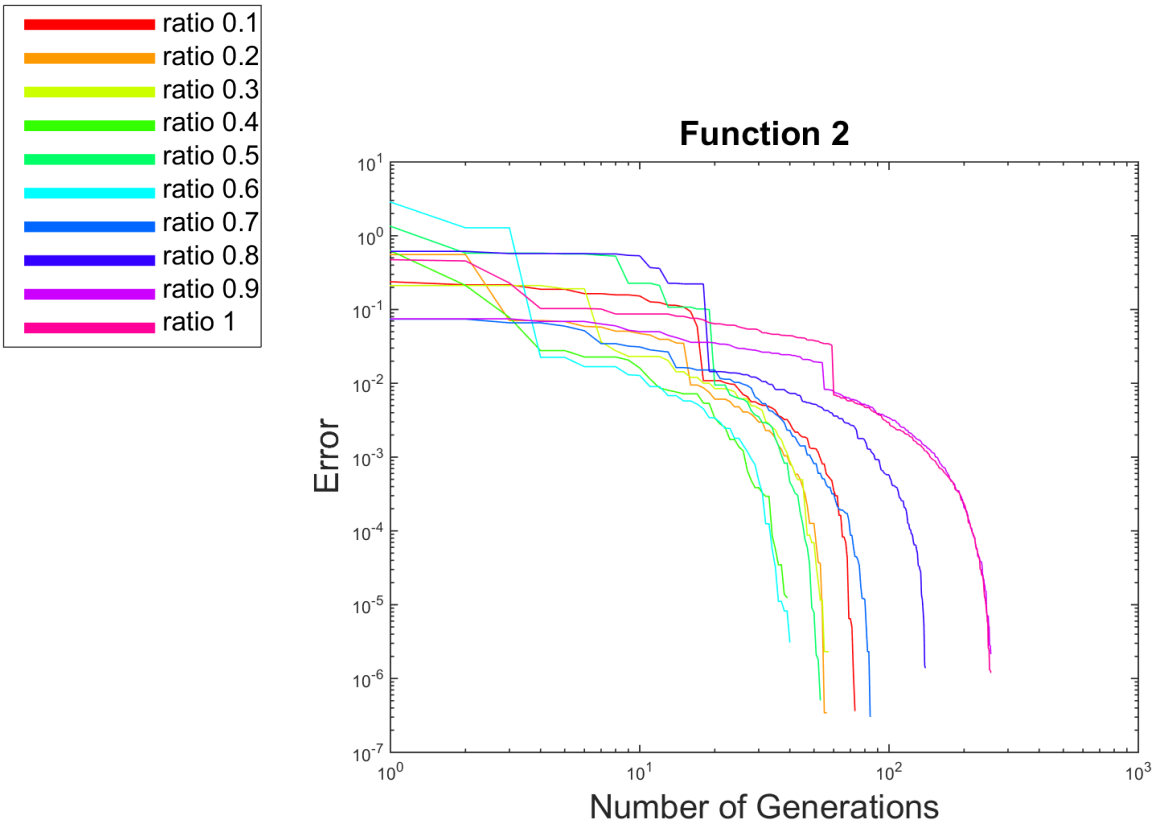


Figure 14.b

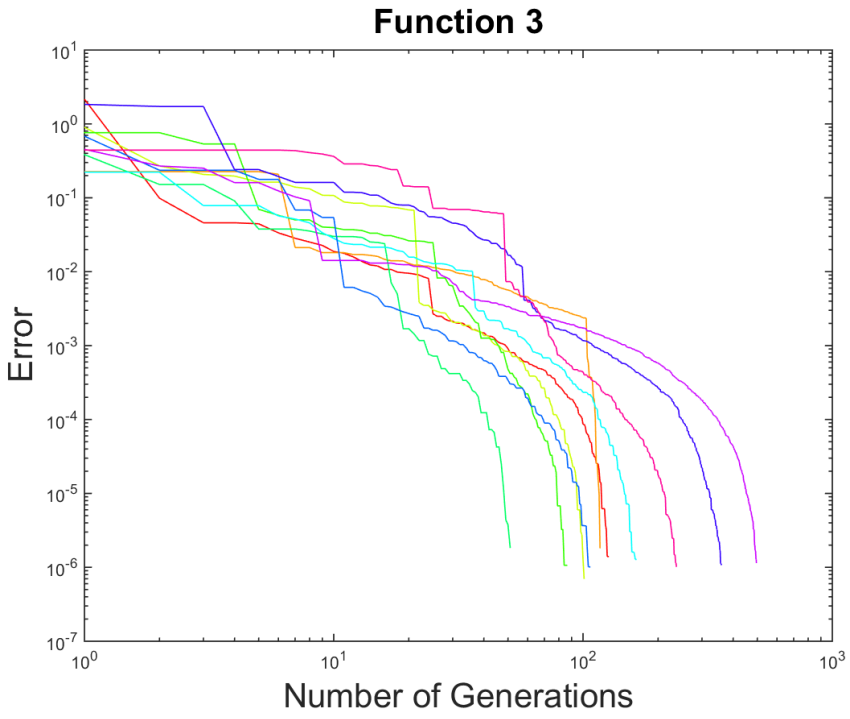


Figure 14.c

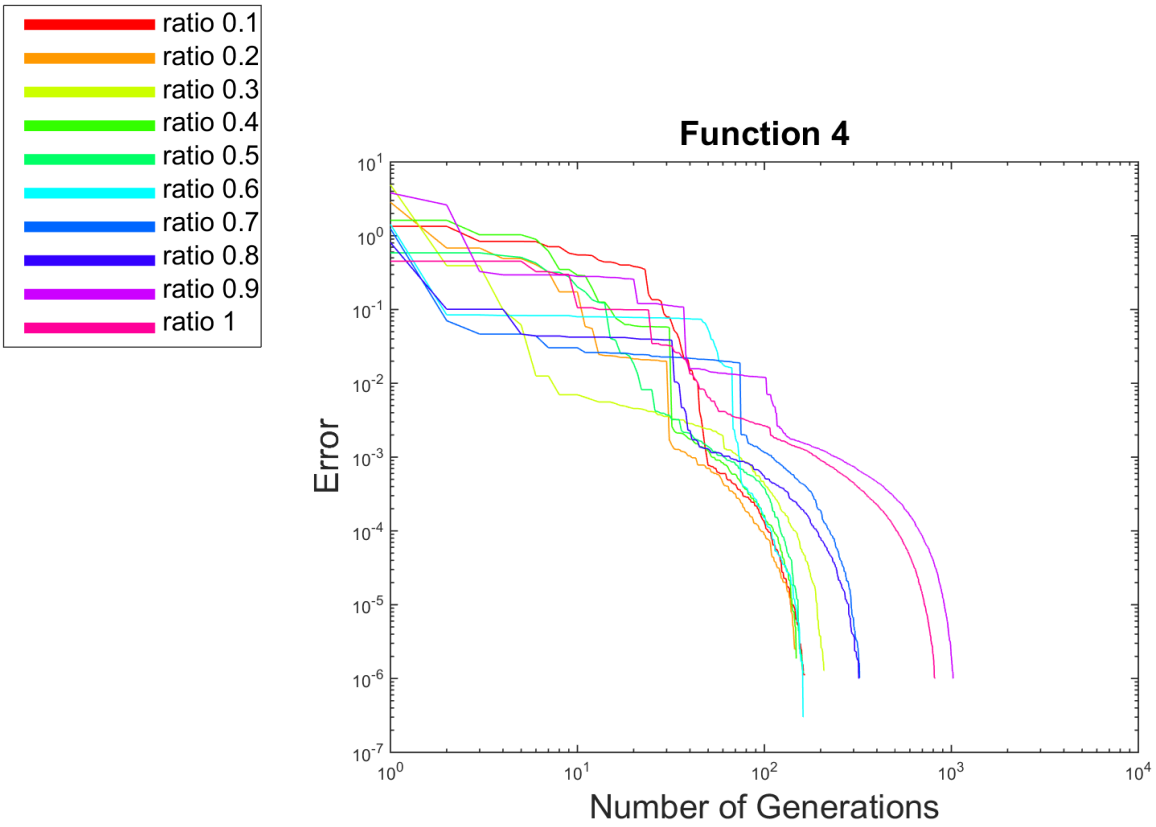


Figure 14.d

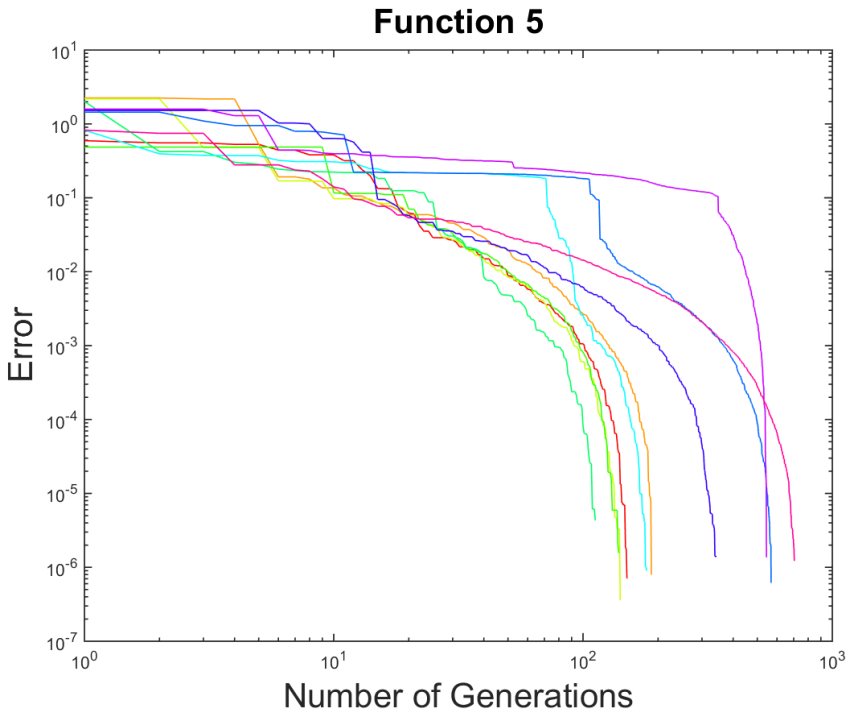


Figure 14.e

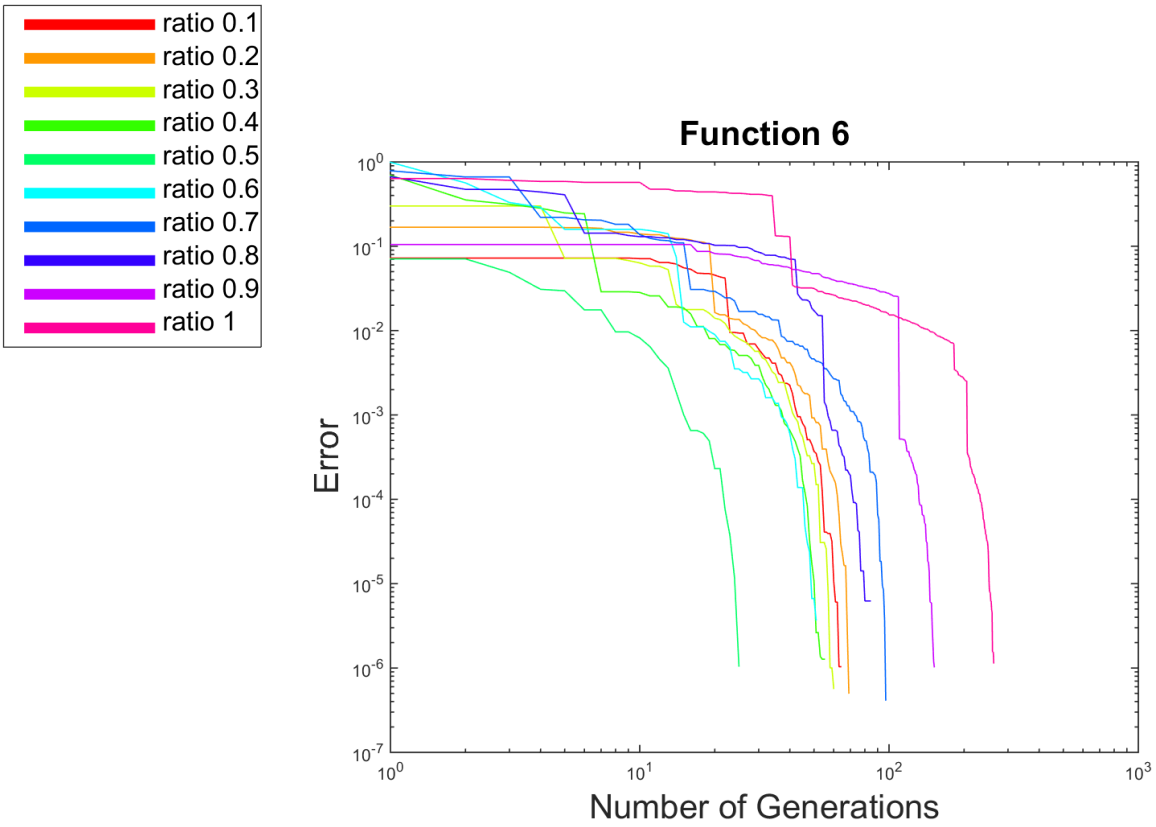


Figure 14.f

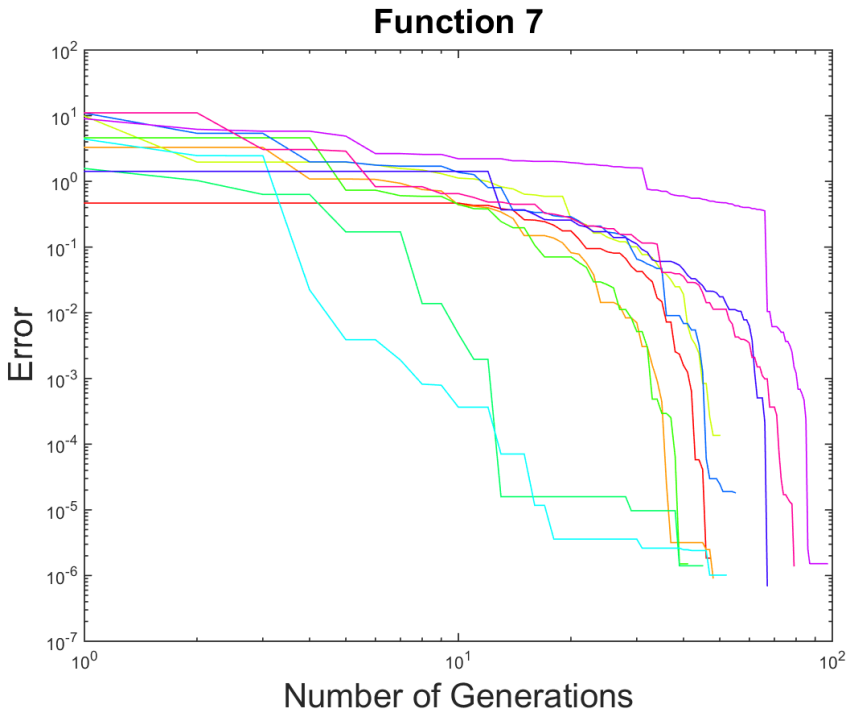


Figure 14.g

Figure 15: Spread in error for adjusting priority given to the latest generation with mutation

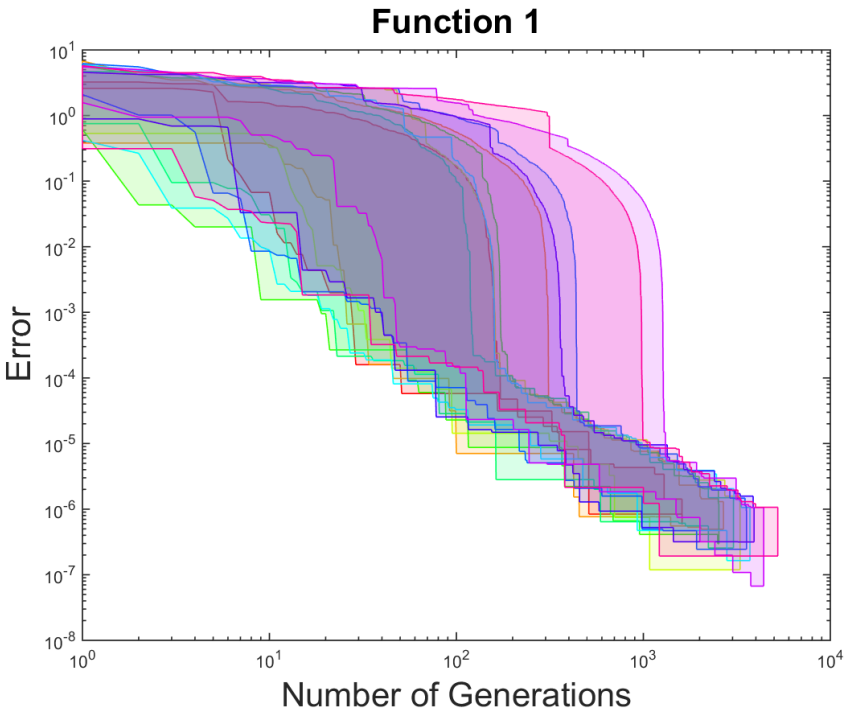
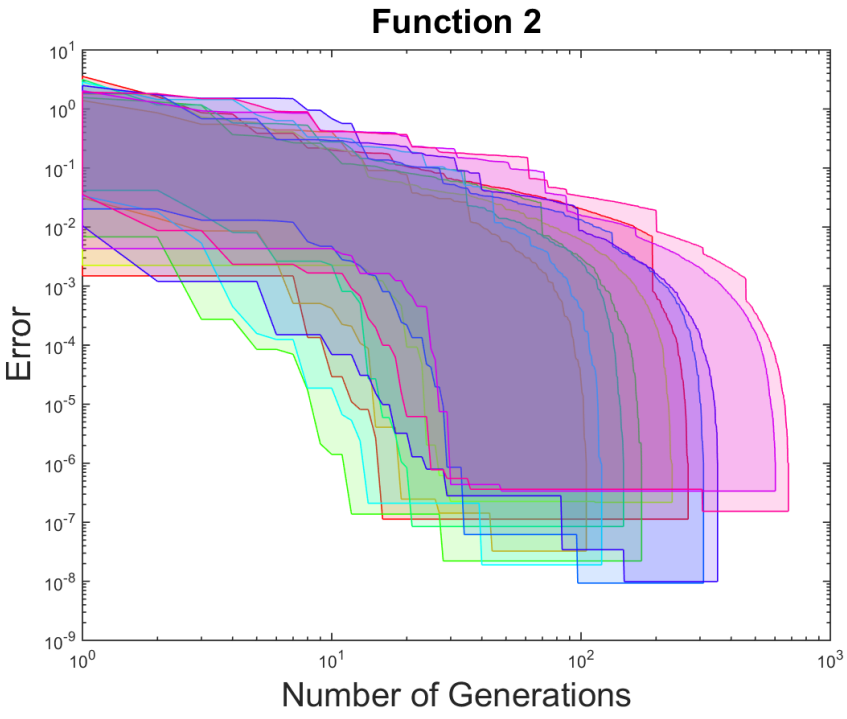


Figure 15.a



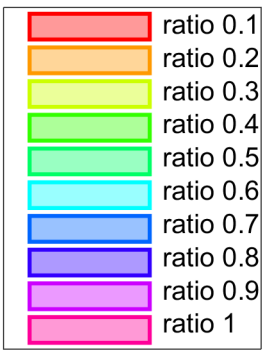


Figure 15.b

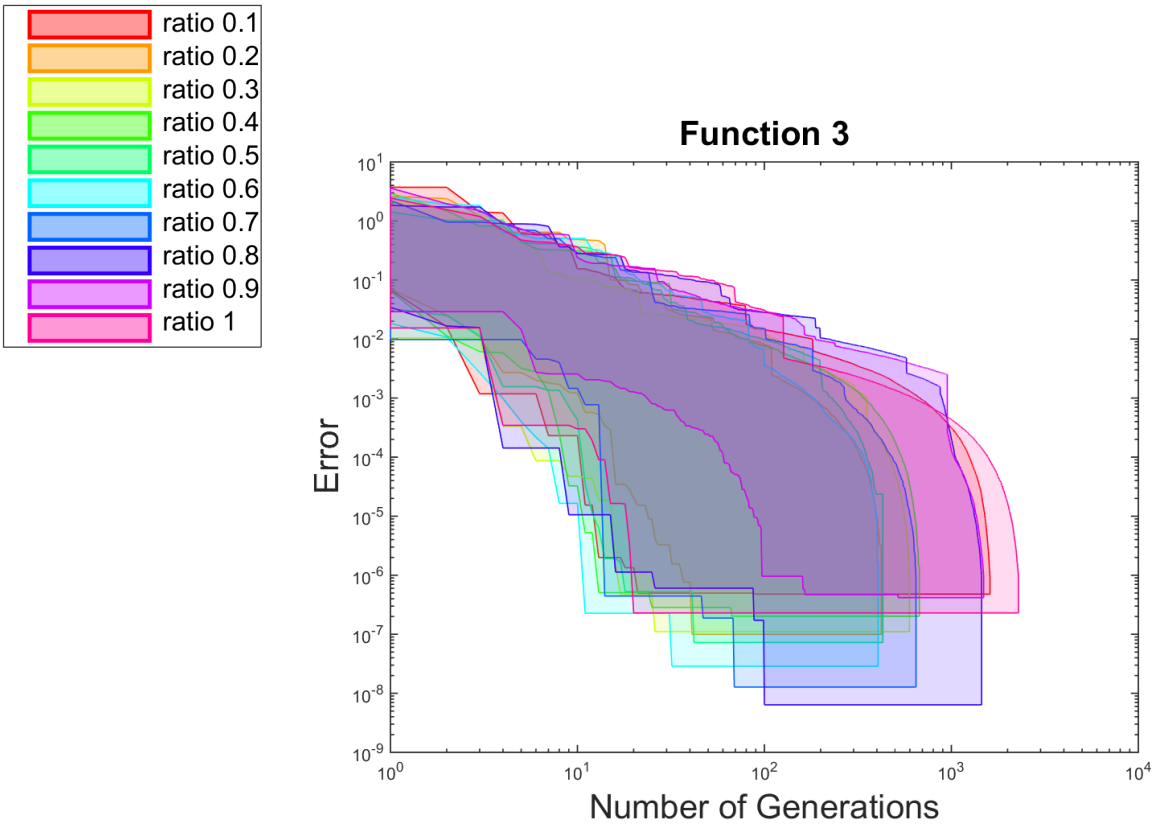


Figure 15.c

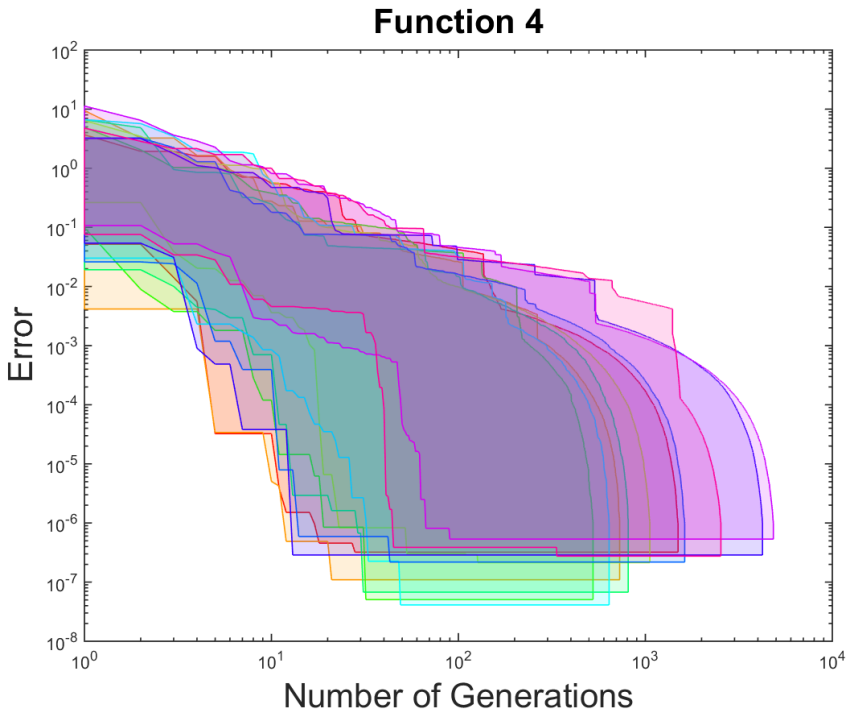


Figure 15.d

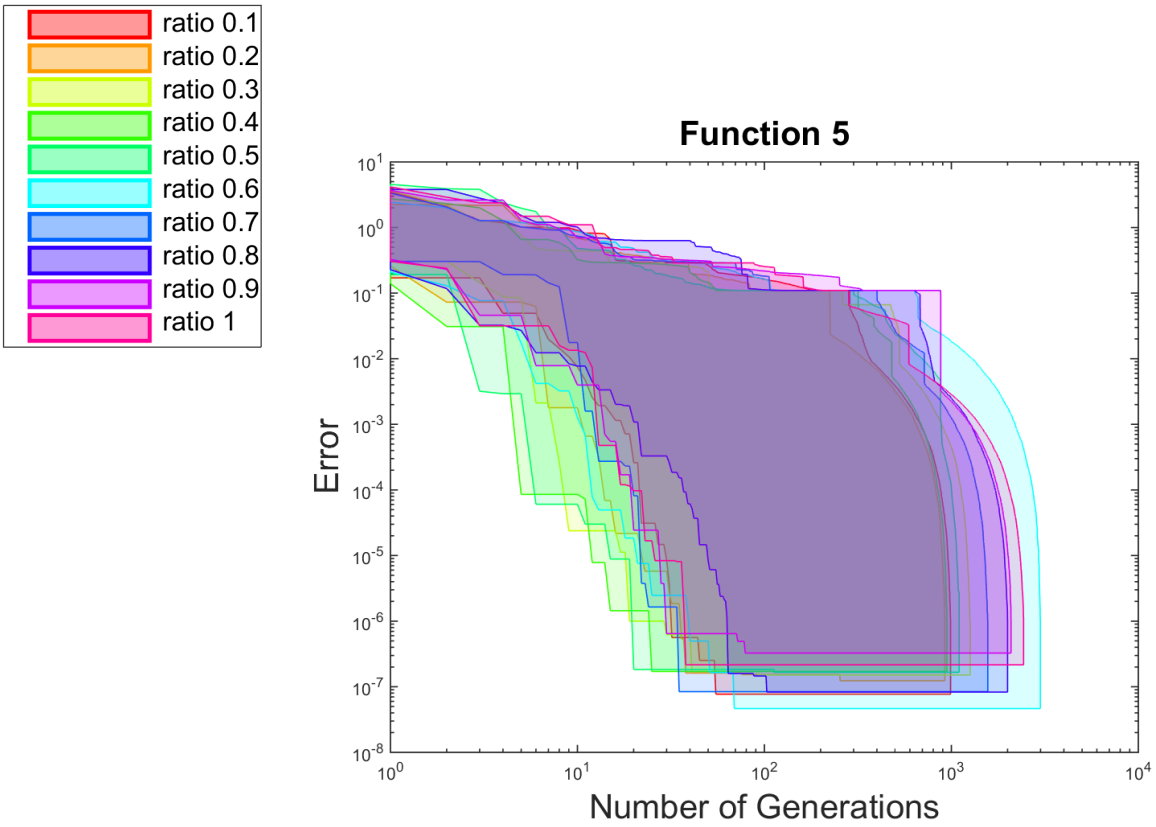


Figure 15.e

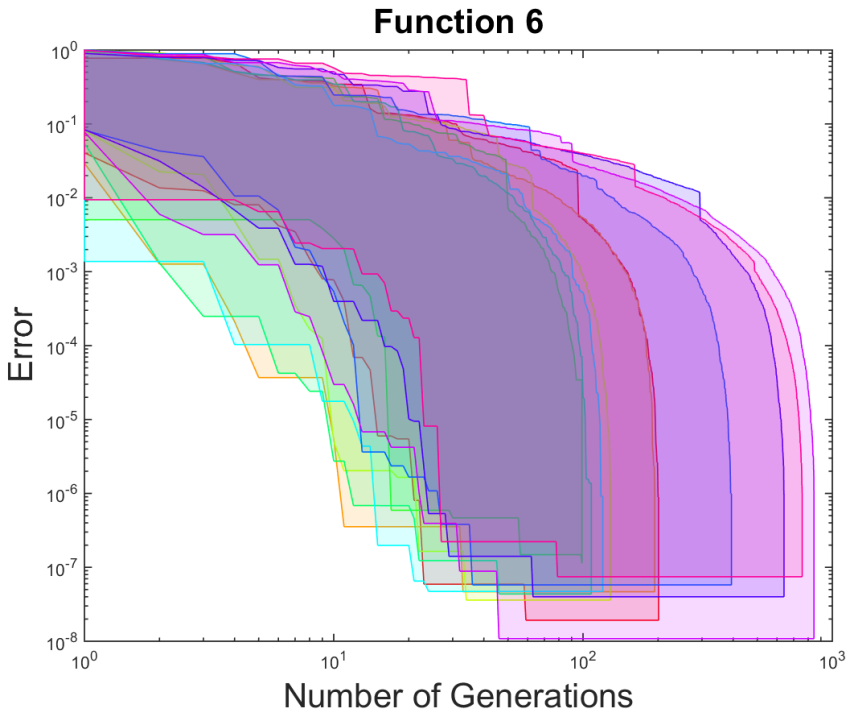


Figure 15.f

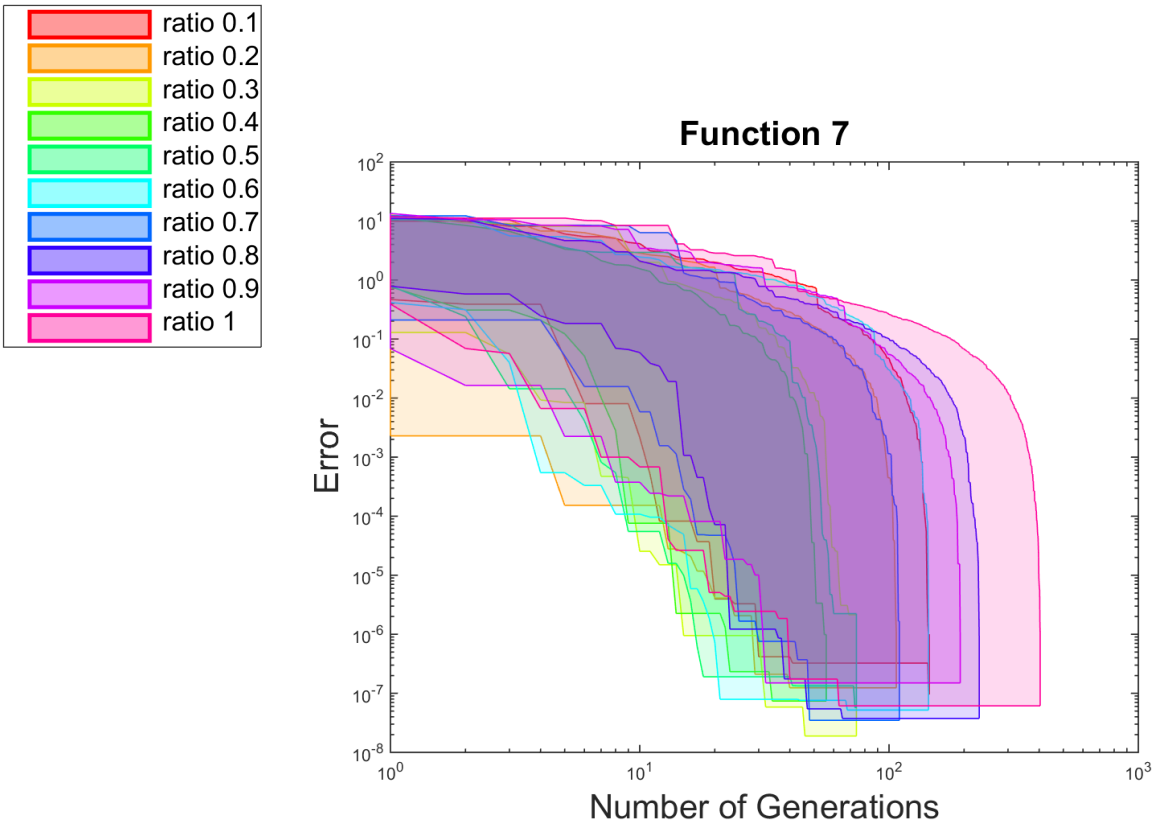


Figure 15.g

- From Figures 14, we can see that from the *medians* of the samples that when the priority probability is between 0.4 and 0.6, we get the faster rates of convergence for the errors.
- For functions 5,6 and 7, we find that the error convergence is not very good as it stays flat for some generations then drops down. This may be due to how the jump function decays as it has an effect on the size of the jumps.
- In Figures 15, from using all the samples and looking at the minimum and the maximum error for each iterations we can see that for most functions, the error varies a lot. This could mean that the data we have got may not give us accurate information or that the priority level does not make a big difference to the error decay.
- For most functions however, we can see that the smallest errors are usually achieved the priority levels between 0.4 and 0.6 and the bigger errors are made by the two ends of the priority spectrum.
- In conclusion, priority levels between 0.4 and 0.6 seem to give the fastest error decay for most functions.

3.4 Number of Elements Needed to Evaluate Before Starting a New Generation - Mutation

Table 5: Parameters used to get the data

Ratio of xBest to the population size	0.6
Ratio of number of elements to evaluate before starting a new generation to the population size	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Number of workers	5
Repeats	20
Population size	50
Priority to the latest generation	0.7
Maximum Time	60 seconds
Tolerance	1×10^{-6}
Functions used	1, 2, 3, 4, 5, 6, 7
Breeding type	Mutation

Figure 16: Results for adjusting the ratio of the number of elements needed to evaluate before starting a new generation to the population size with mutation

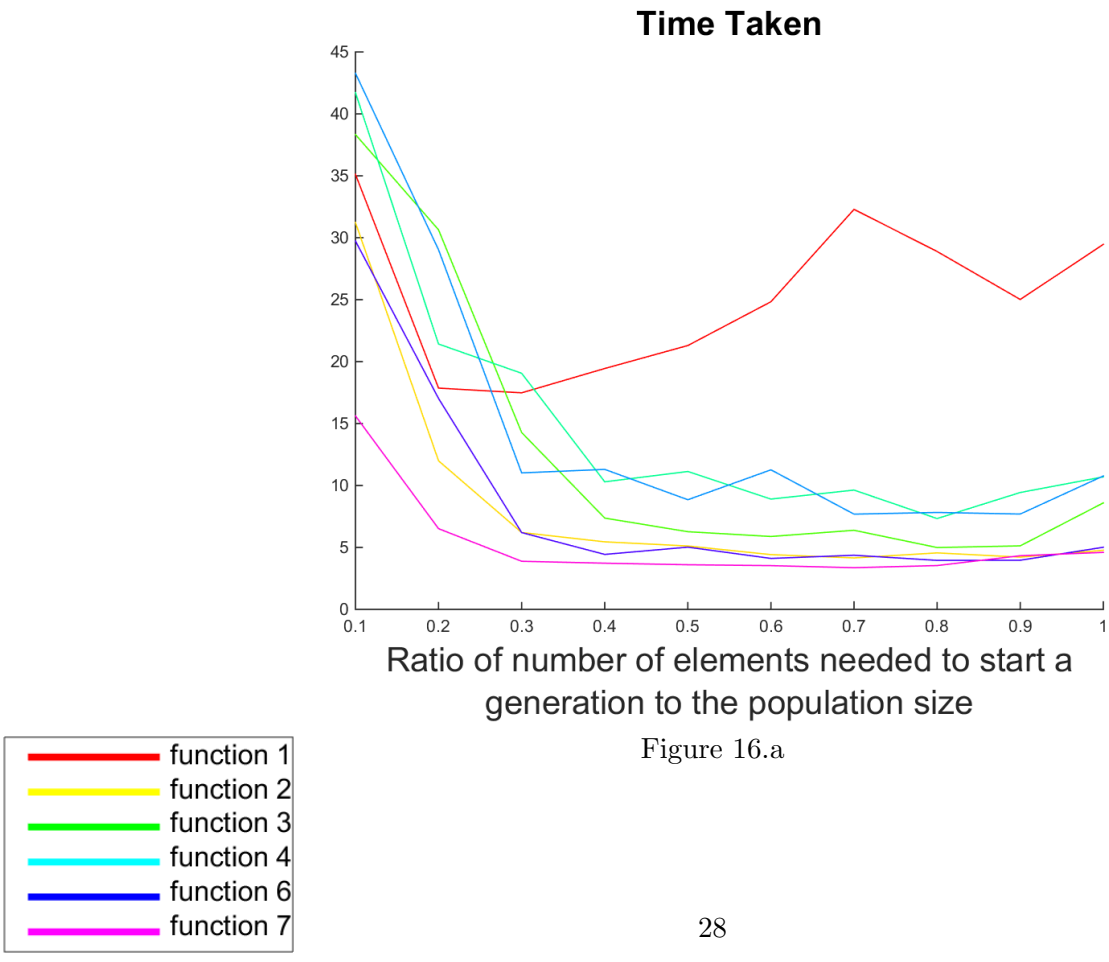


Figure 16.a

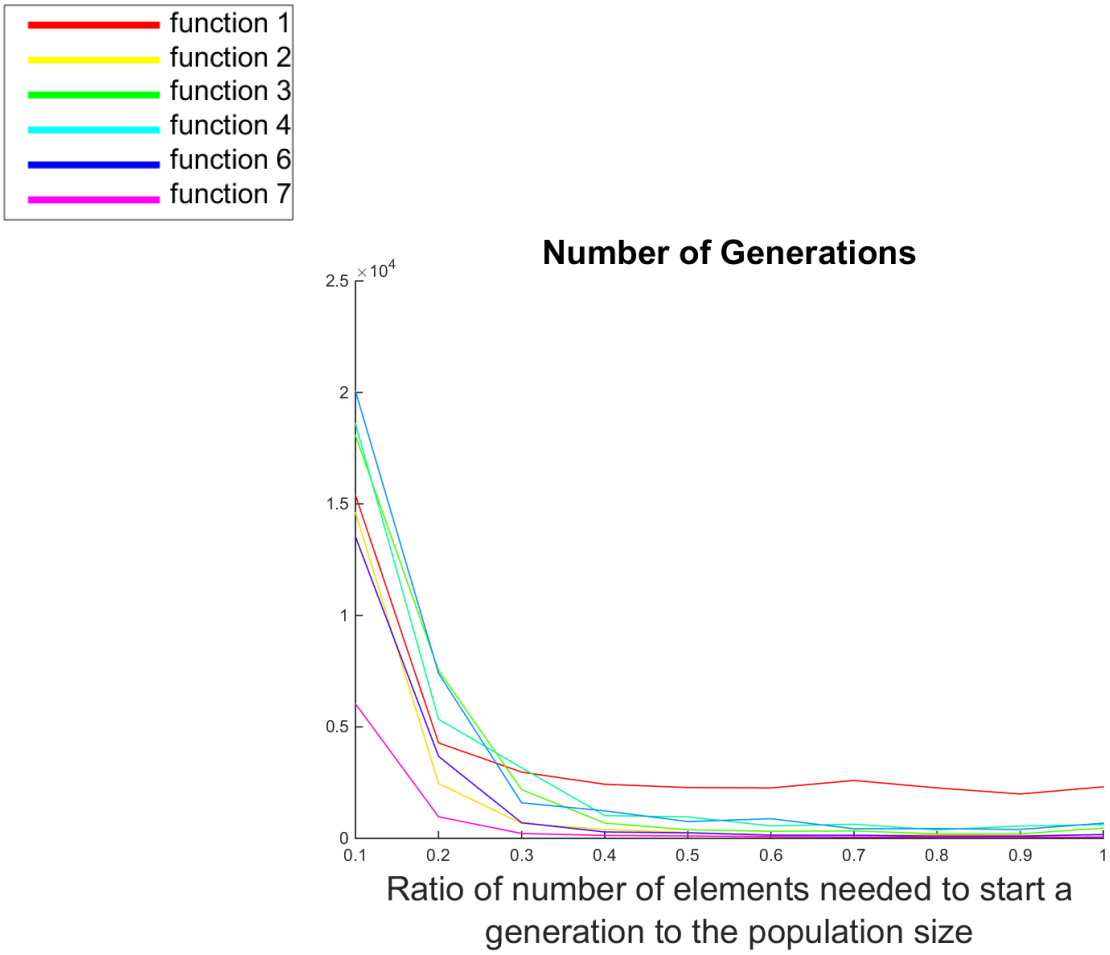


Figure 16.b

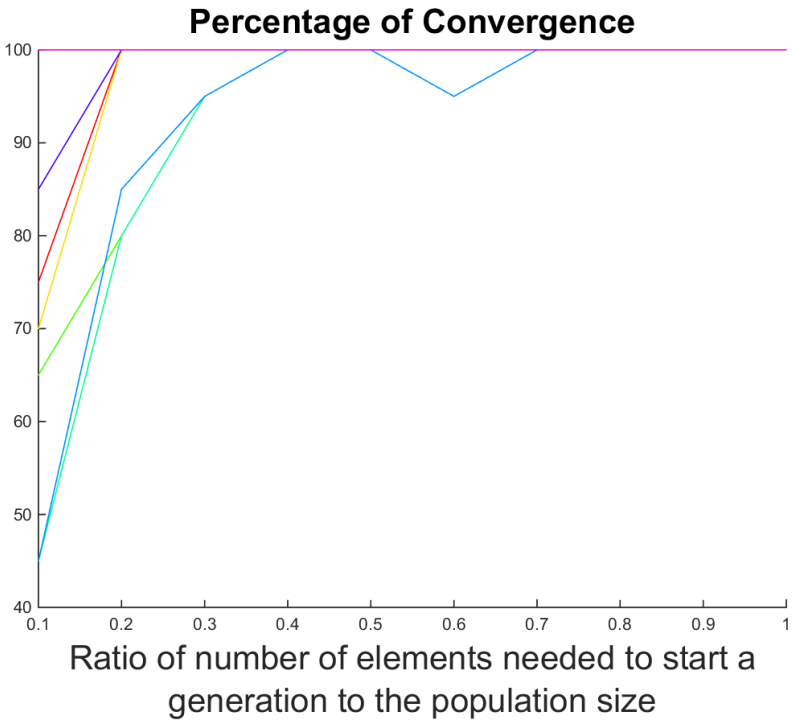


Figure 16.c

- From Figure 16.c, we can see that when the ratio in below 0.4, the percentage of convergence is a mostly below 100%. From Figure 16.a, we can see that between ratio of 0.4 and 0.9, the time take for convergence is mostly flat except for function 1. Between ratio of 0.9 and 1, we see an increase in time, this indicates that the extra evaluations did not give us any better information and took a longer time.
- Figure 16.b shows that as the ratio increases, the number of generations decreases to a certain point and is then mostly flat. This is because we get more information from one generation hence need fewer generations to reach the minimum.
- One point we need to take into account is that the function we are evaluating here takes little time to evaluate however the function evaluation might take a lot more time, so as we increase the ratio, the more evaluations would be made per generation which would take longer. Taking this into account the best range for the ratio would be between 0.4 and 0.6, to keep the time taken from evaluations low.
- In conclusion, for mutation, ratios between 0.4 and 0.6 give the best results.

Figure 17: Errors for adjusting the ratio of the number of elements needed to evaluate before starting a new generation to the population size with mutation

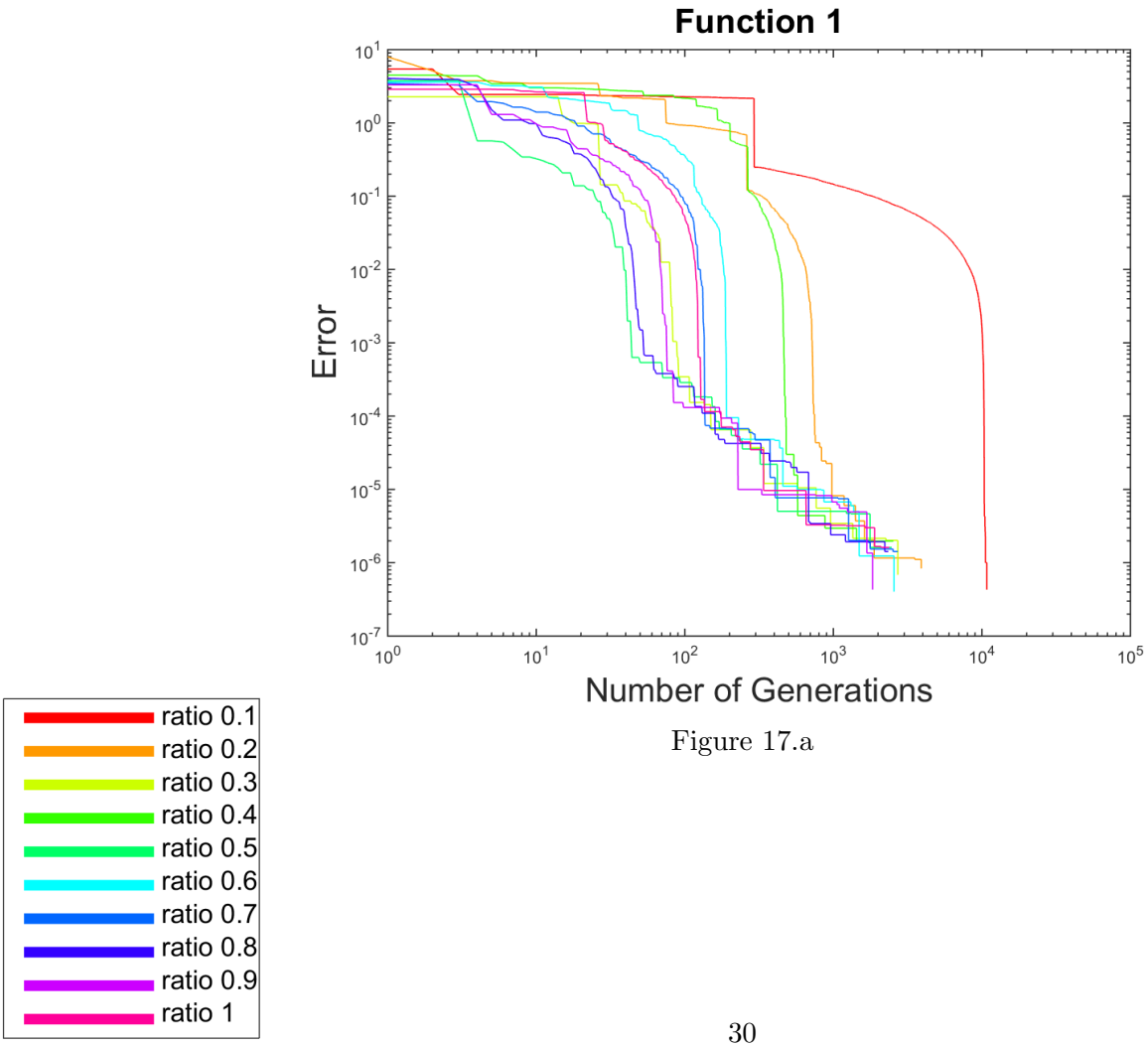


Figure 17.a

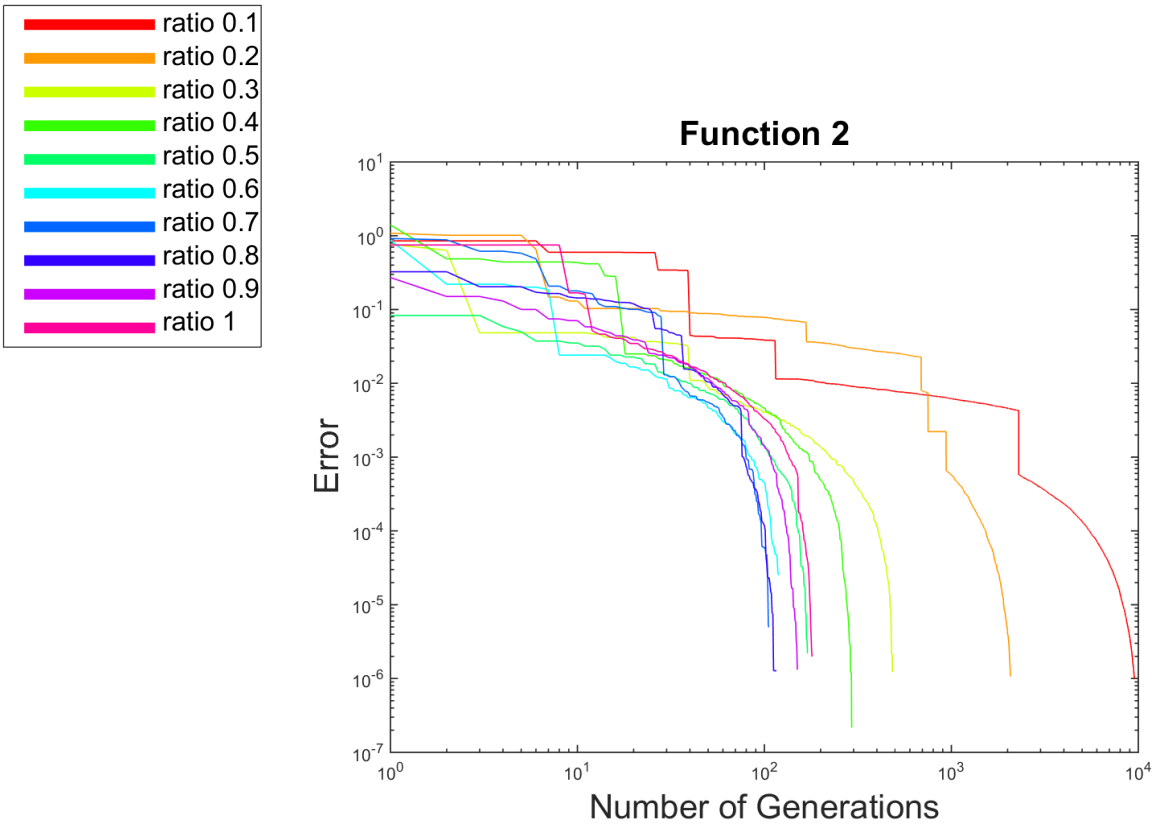


Figure 17.b

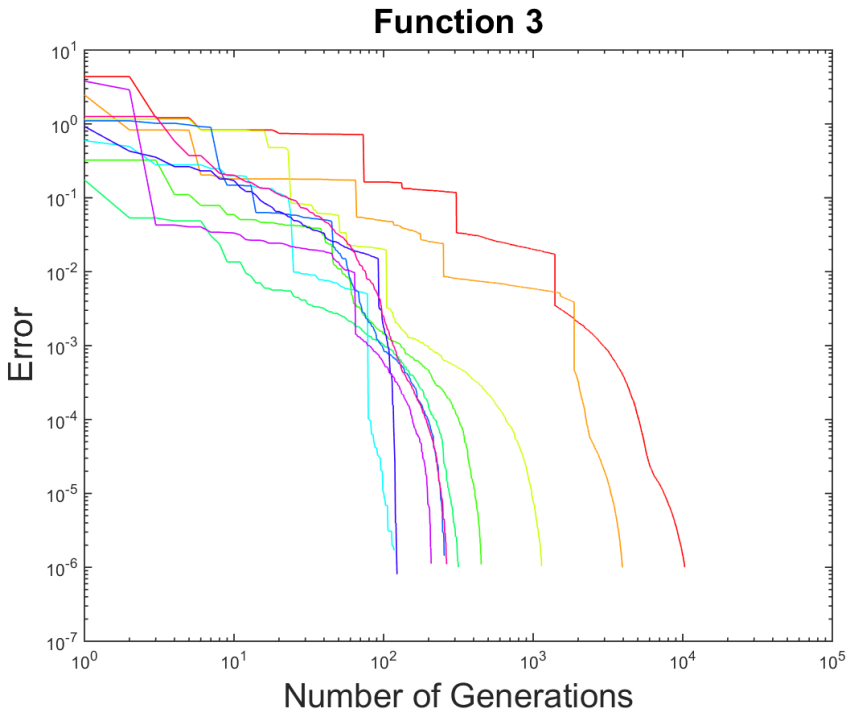


Figure 17.c

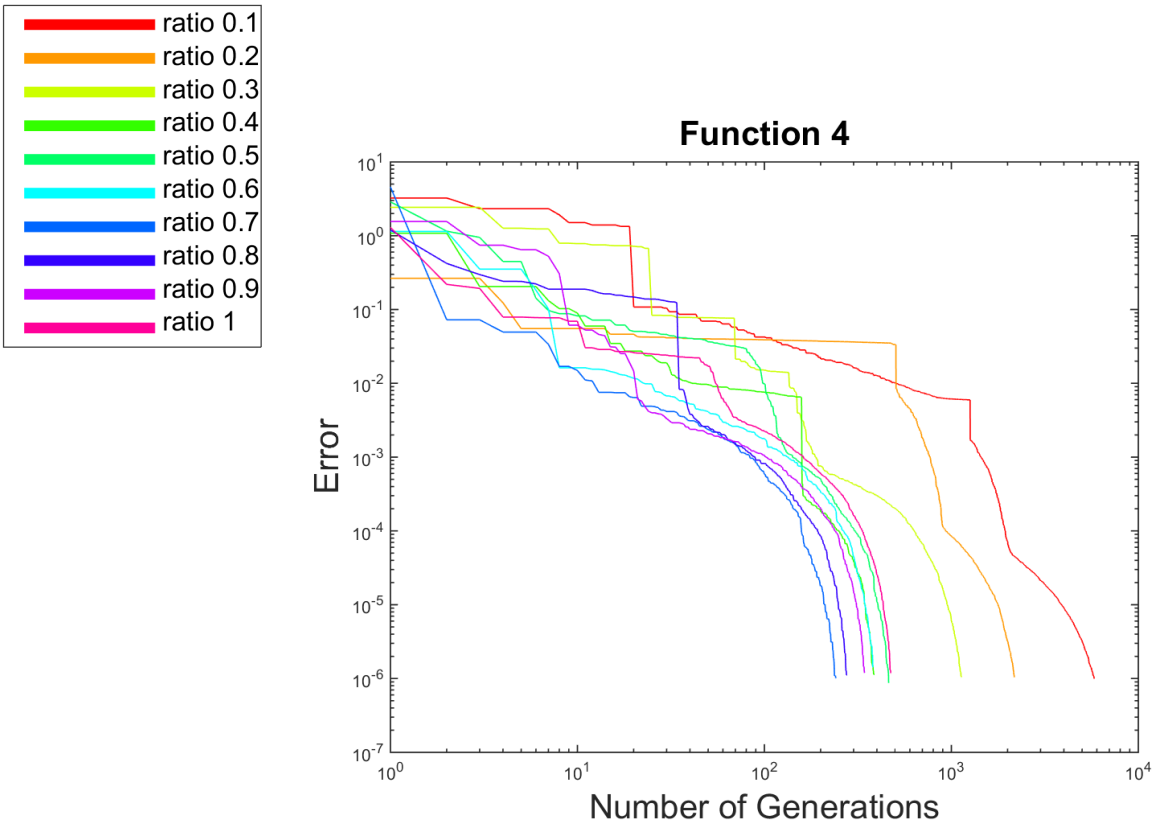


Figure 17.d

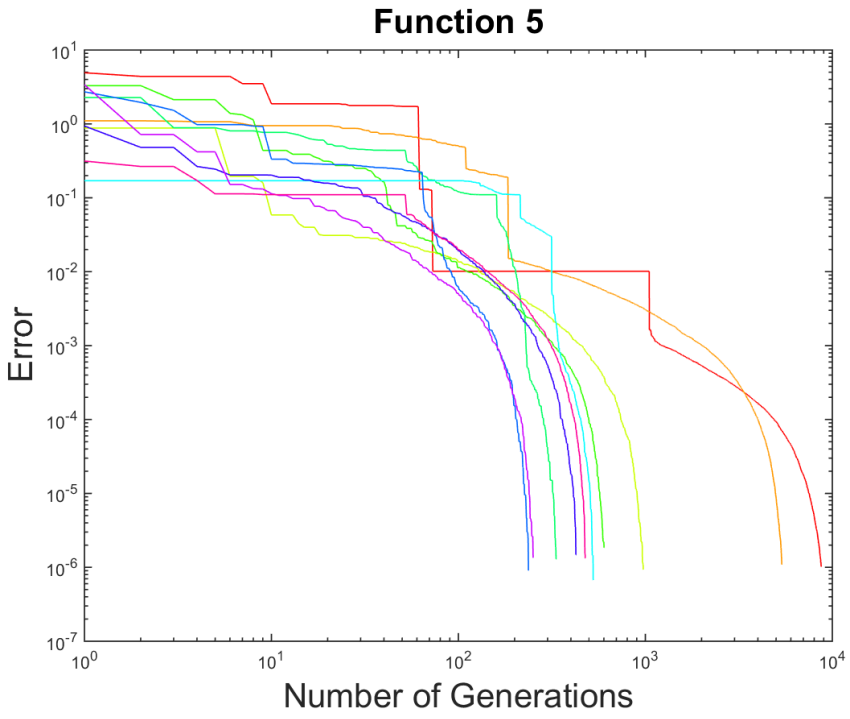


Figure 17.e

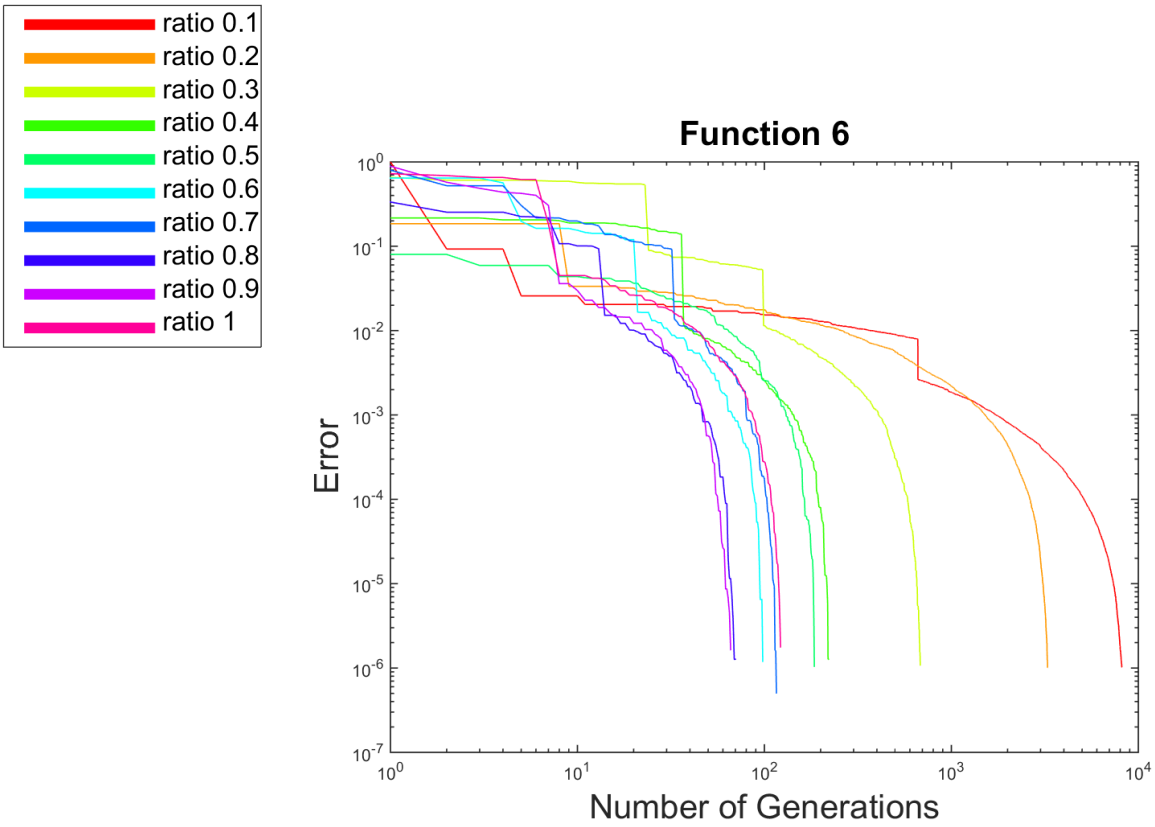


Figure 17.f

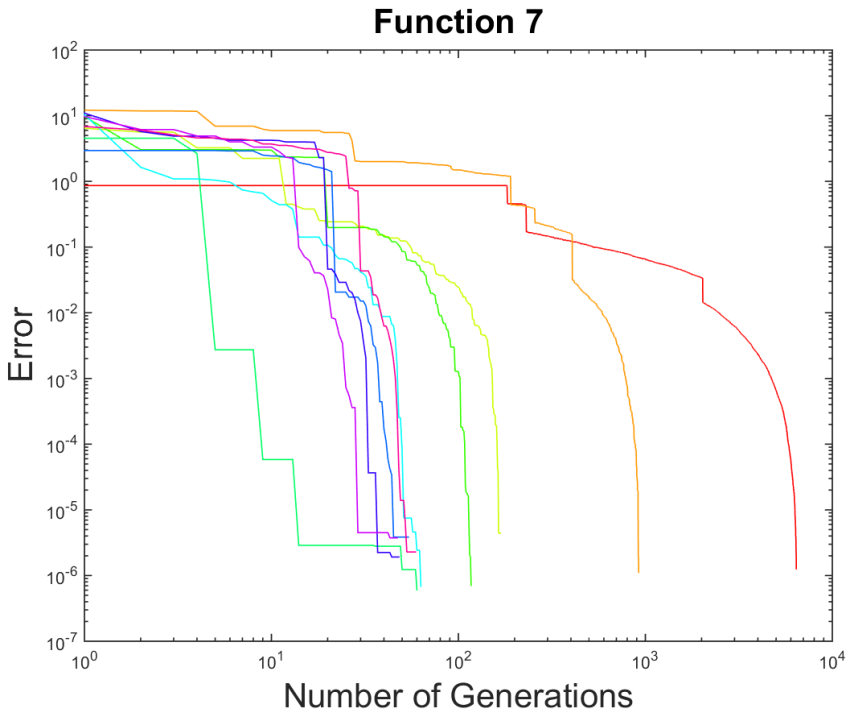


Figure 17.g

Figure 18: Spread in error for adjusting the ratio of the number of elements needed to evaluate before starting a new generation to the population size with mutation

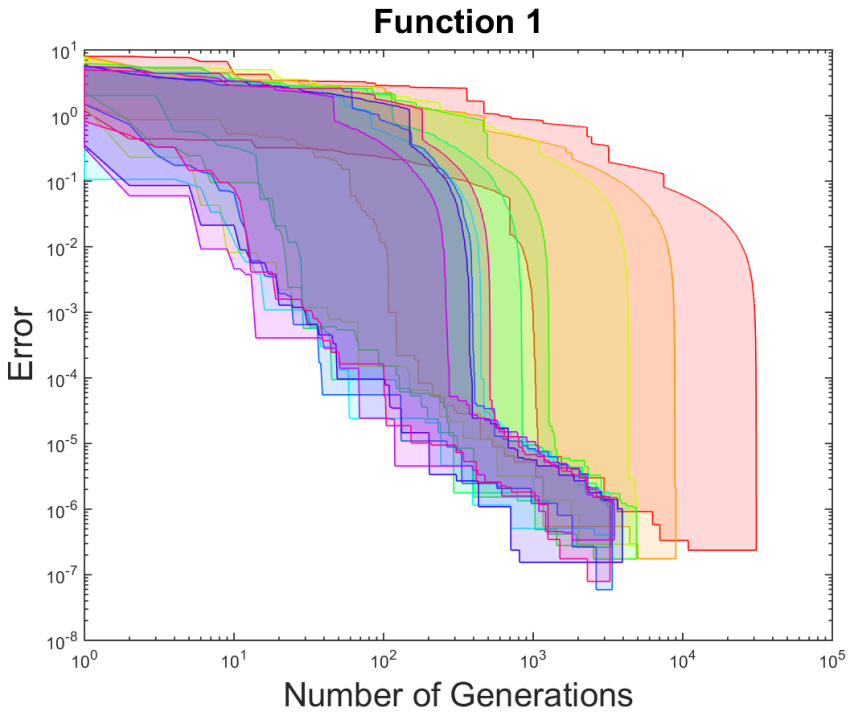
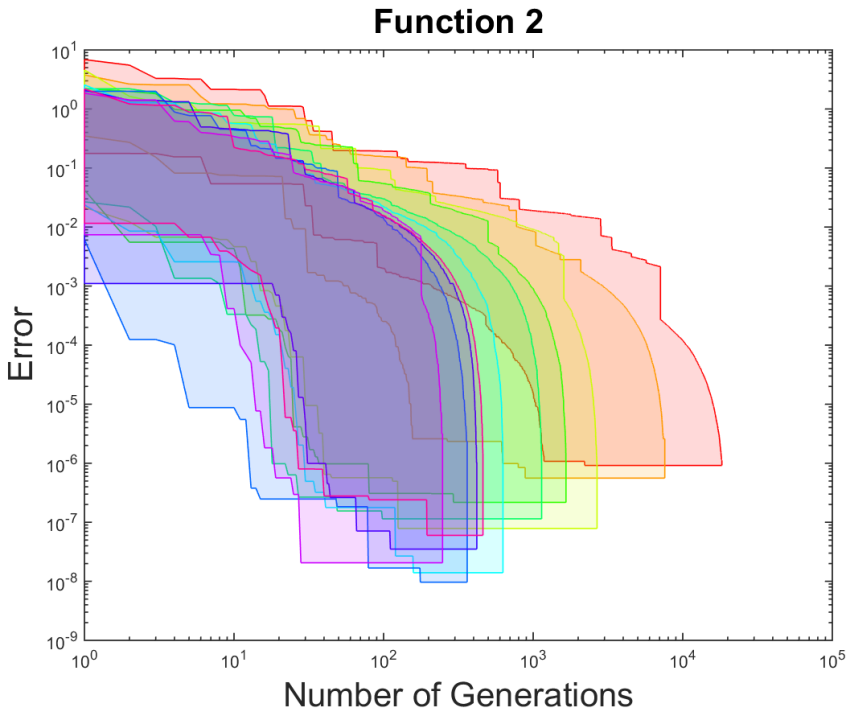


Figure 18.a



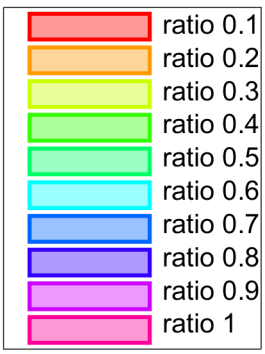


Figure 18.b

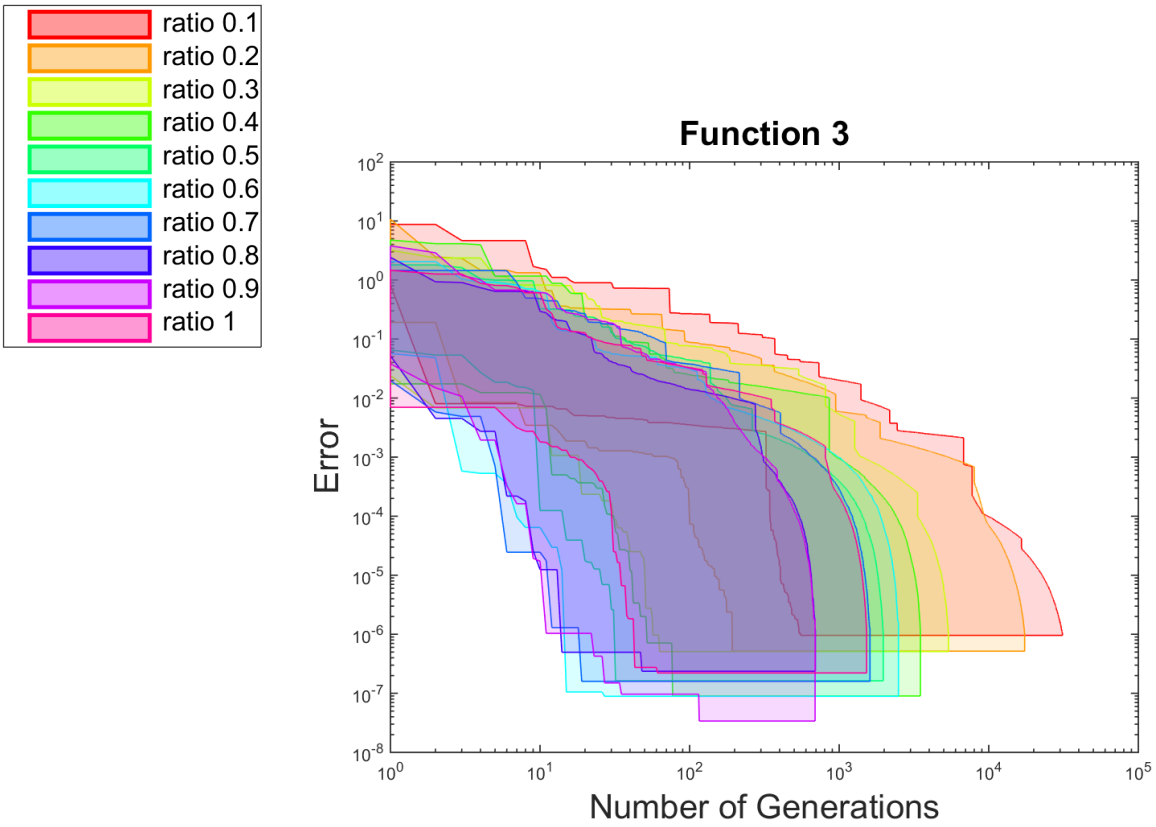


Figure 18.c

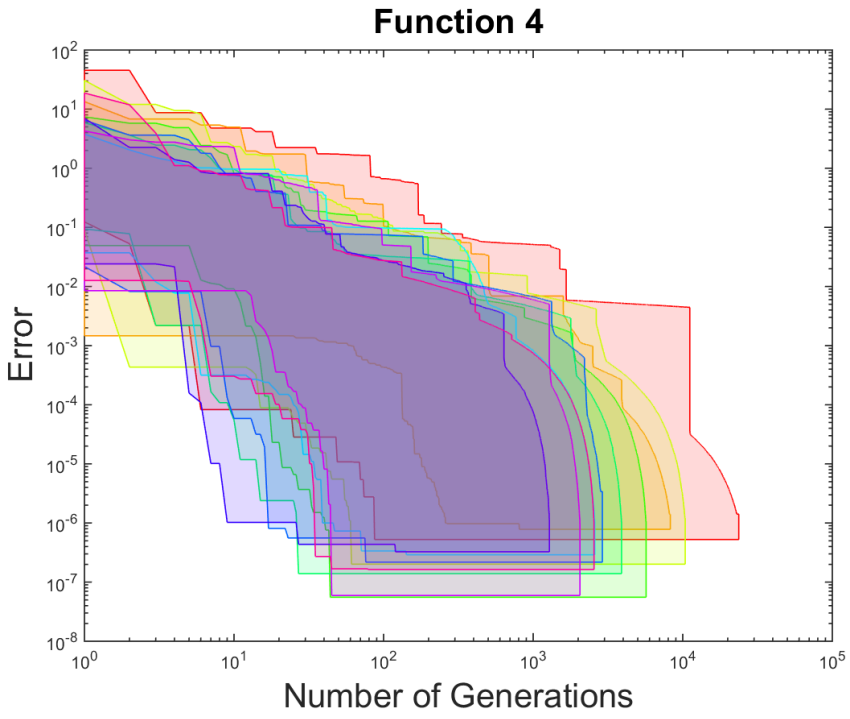


Figure 18.d

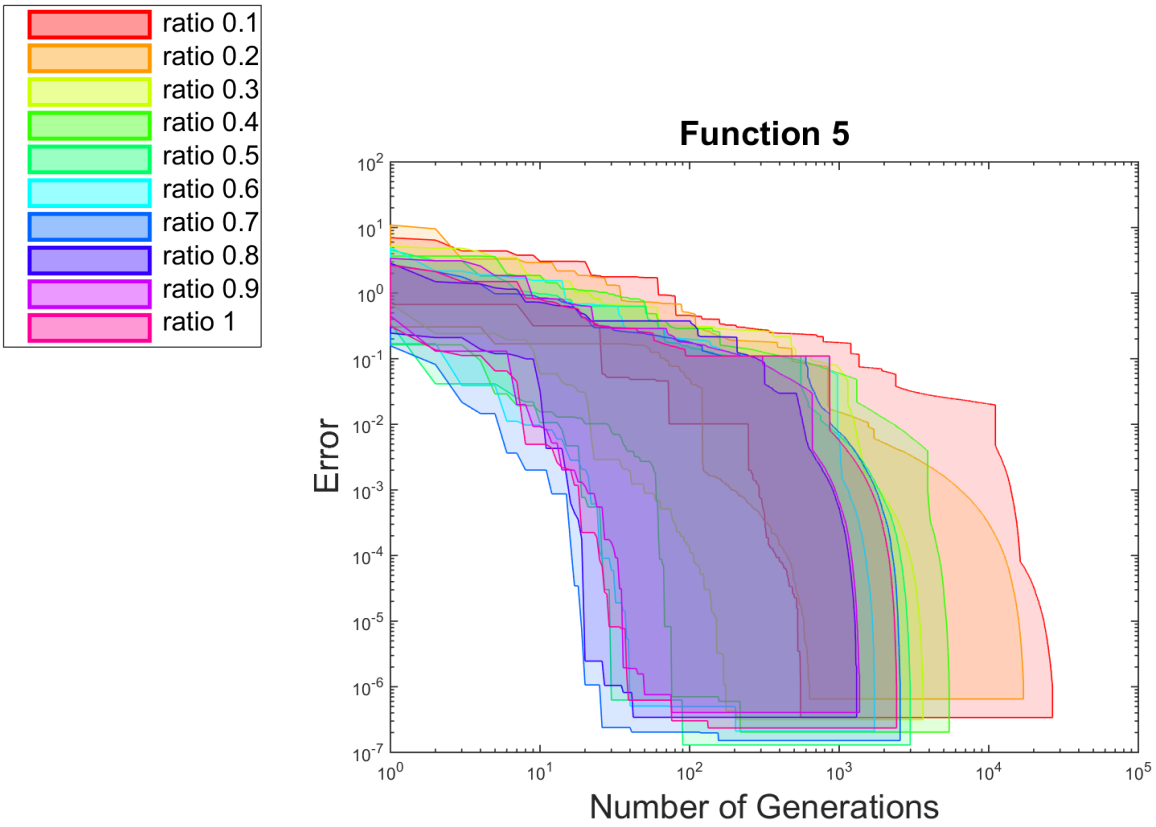


Figure 18.e

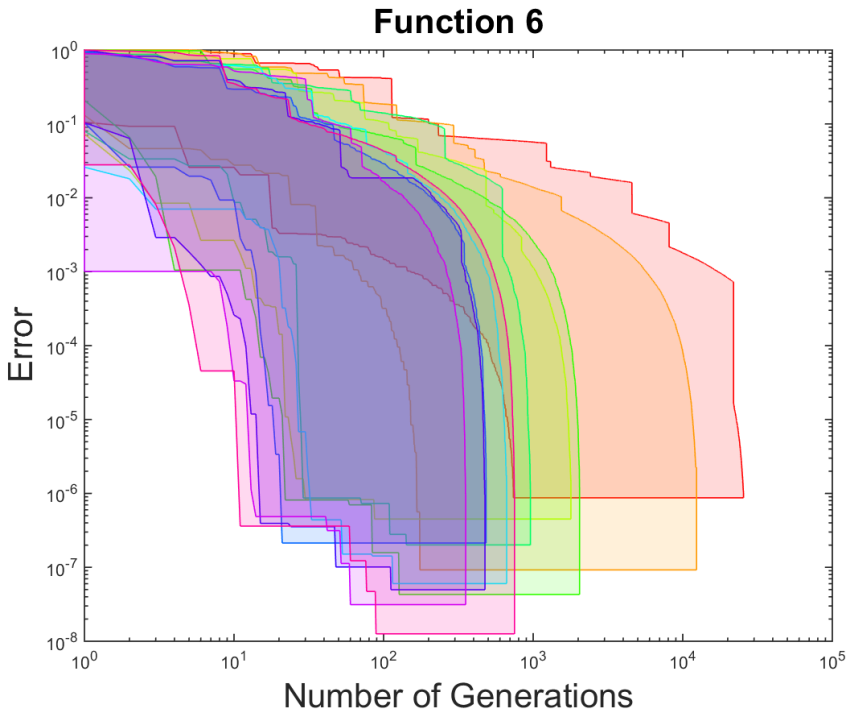


Figure 18.f

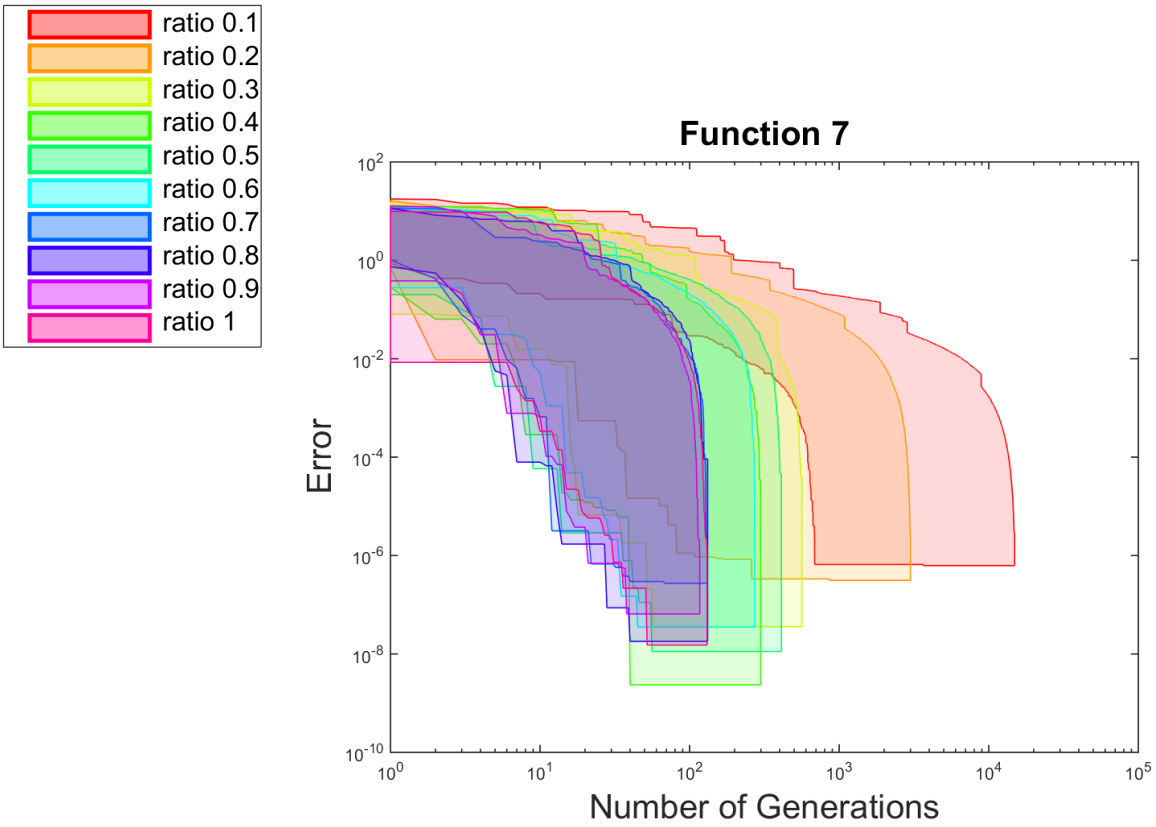


Figure 18.g

- From Figures 17, we see that ratio of 0.1 and 0.2 for every function gives the slowest decay of error.
- For ratios between 0.4 and 1, we seem to get quite similar error decays for most functions.
- From Figures 18, we can see that the spread is a lot less than what we saw in Section 3.3. This indicates that the results are more accurate therefore more informative.
- Similar to as seen in the Figures 17, ratio of 0.1 has the slowest error decay by far. We see that the spreads of most of the ratios between 0.4 and 1 overlap each other a lot which means they all have fairly similar error decays.
- In conclusion, we see that low ratio has the slowest error decay and ratios from 0.4 to 1 give faster and similar error decays.

3.5 Size of xBest - Mutation

Table 6: Parameters used to get the data

Ratio of xBest to the population size	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Ratio of number of elements to evaluate before starting a new generation to the population size	0.6
Number of workers	5
Repeats	20
Population size	50
Priority to the latest generation	0.7
Maximum Time	60 seconds
Tolerance	1×10^{-6}
Functions used	1, 2, 3, 4, 5, 6, 7
Breeding type	Mutation

Figure 19: Results for adjusting the ratio of the size of xBest to the population size with mutation

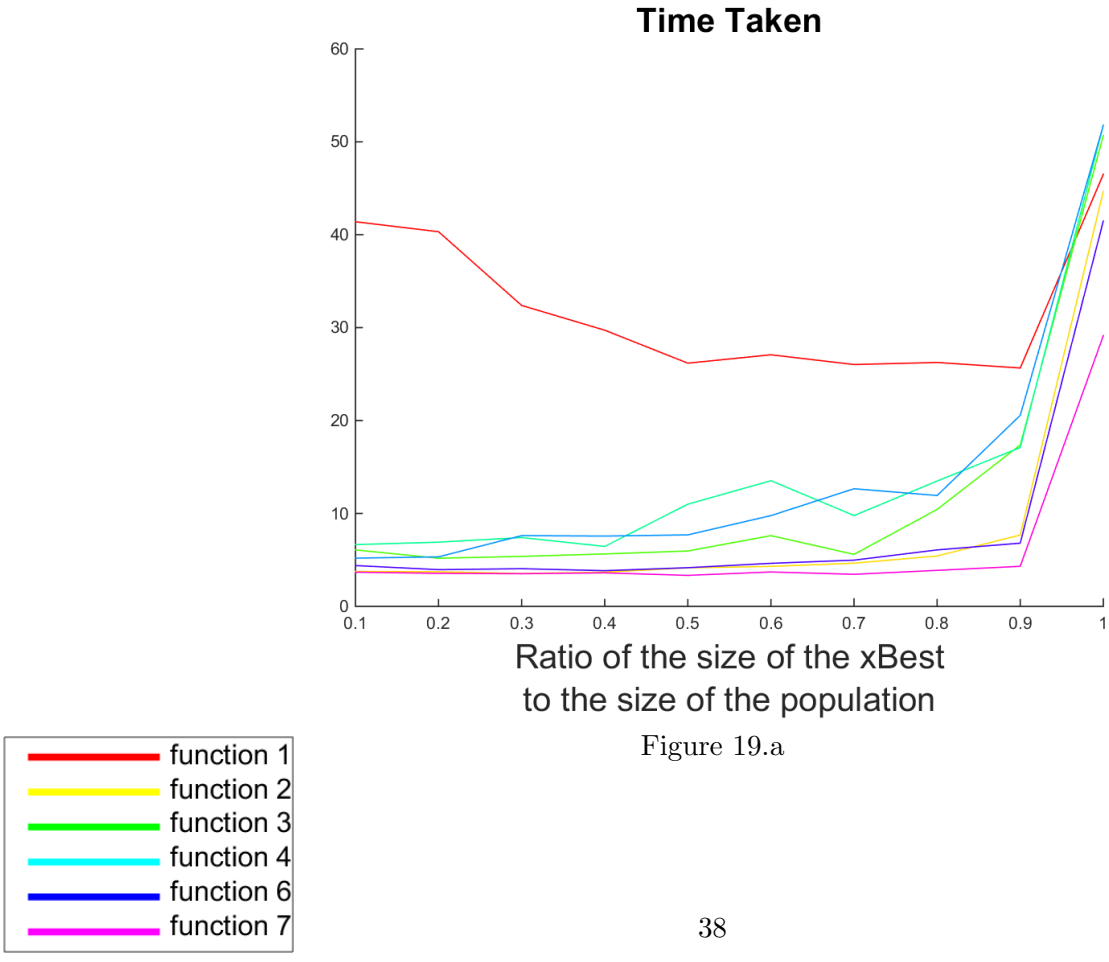


Figure 19.a

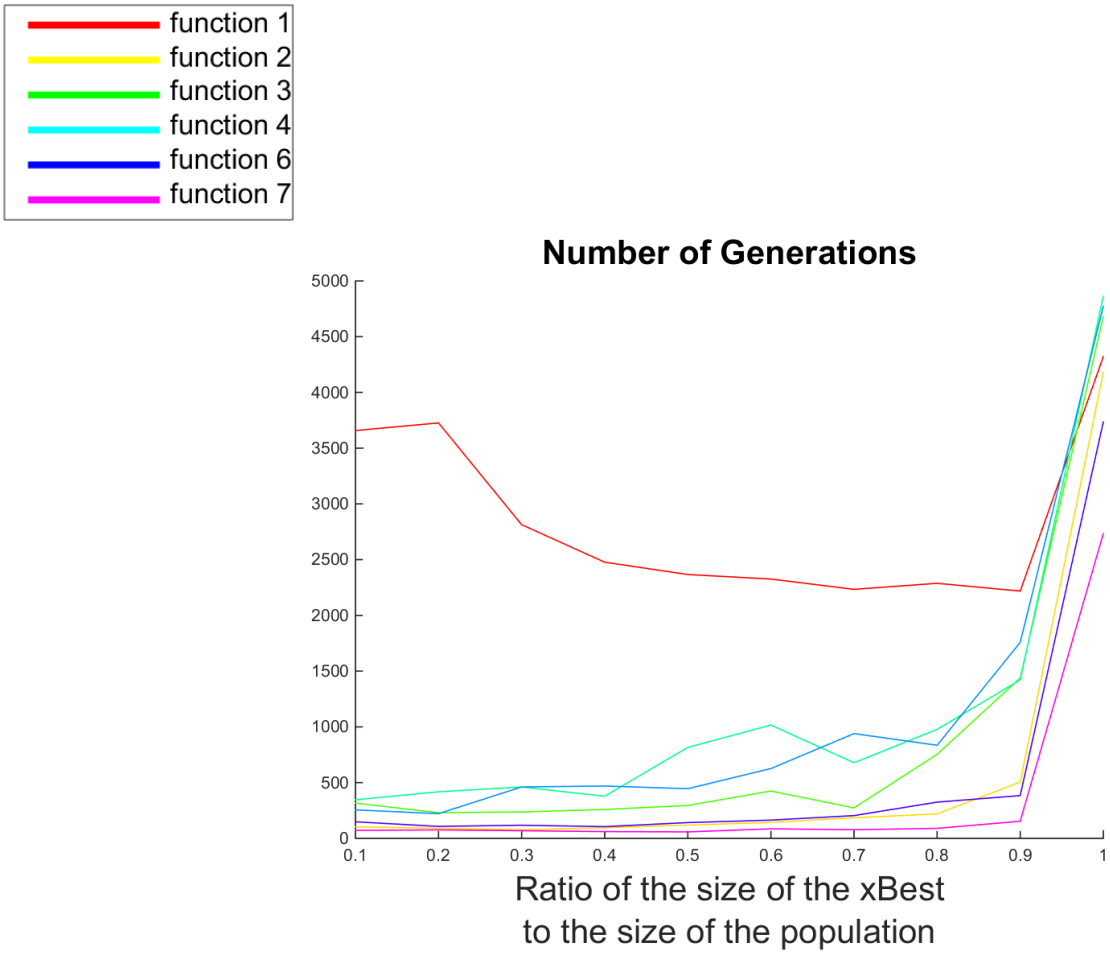


Figure 19.b

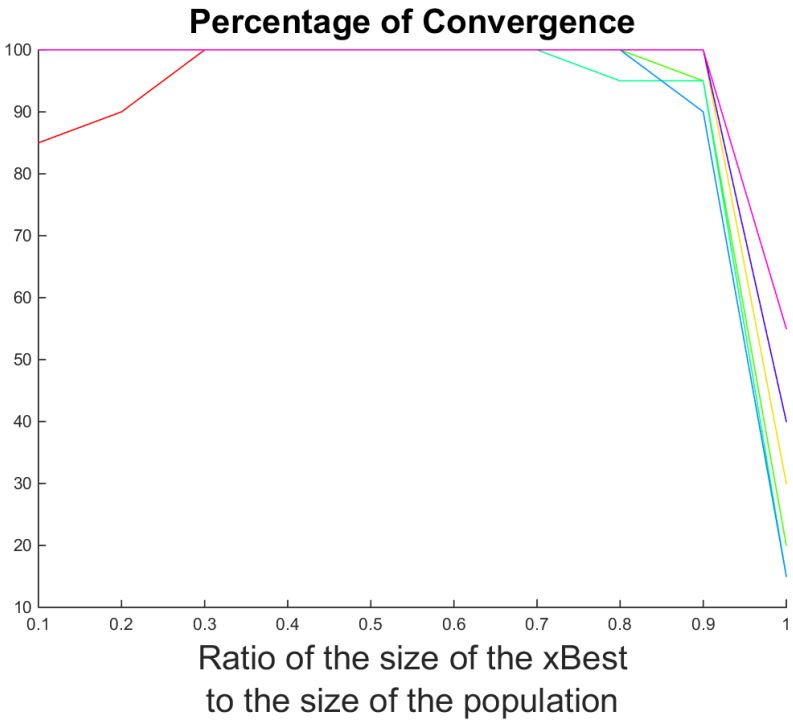


Figure 19.c

- From Figures 19.a and 19.b, we see that the time taken is proportional to the number of generations reached and this is because the number of elements to evaluate because starting a new generation is fixed.
- We see that for most functions the time taken is flat up till ratio 0.5 then time taken starts to increase. However, for function 1, the time decreases until ratio 0.5 then is flat until 0.9 then it increases for 1. This tells us that have the ratio equal to 1 is bad because there are no random points that form the population as it is completely made from the xBest.
- Figure 19.c shows all functions converge for ratio between 0.3 and 0.7.
- In conclusion, for mutation, the ratio of 0.5 gives the fastest convergence to the minimum.

Figure 20: Errors for adjusting the ratio of the size of xBest to the population size with mutation

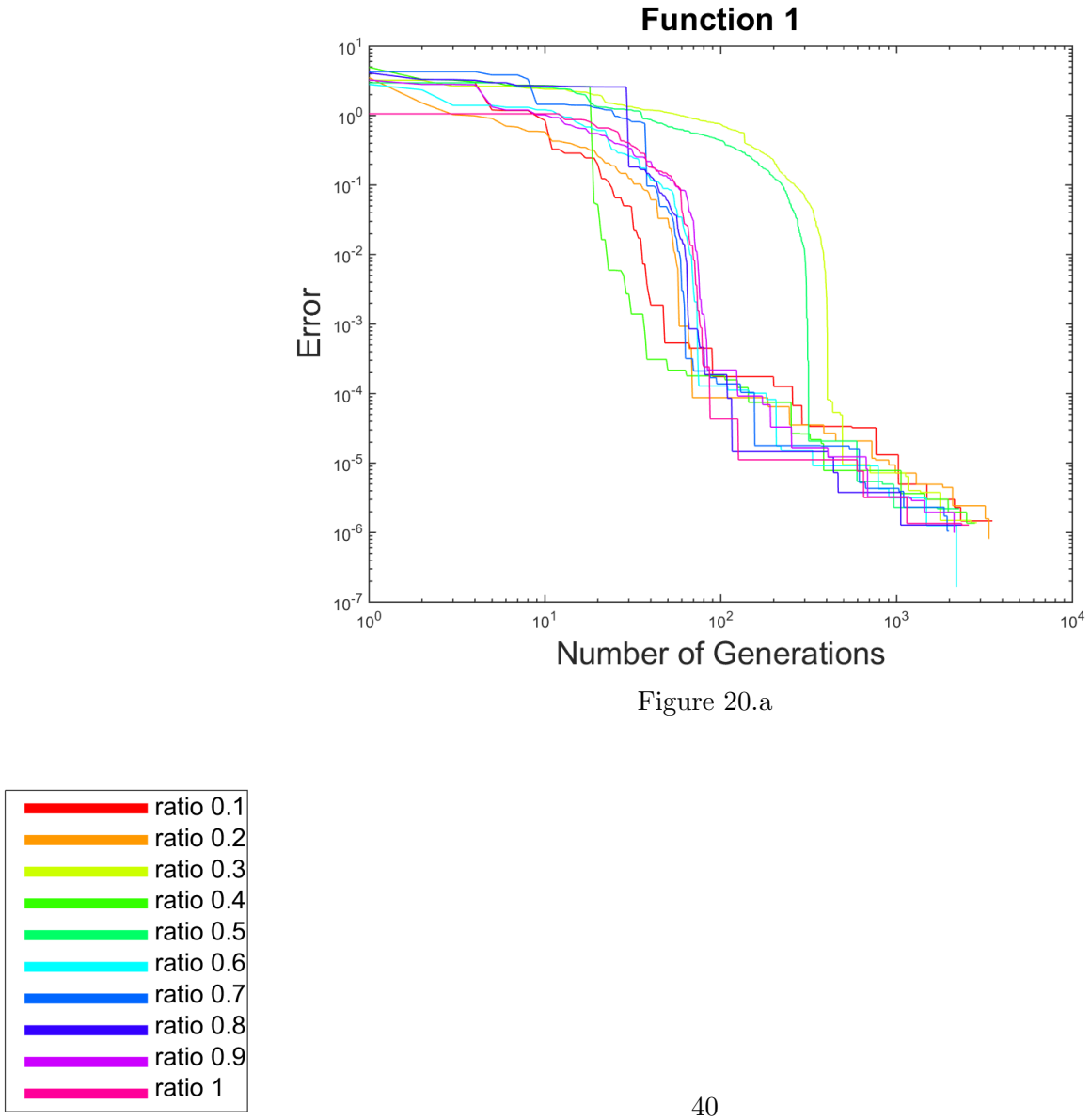


Figure 20.a

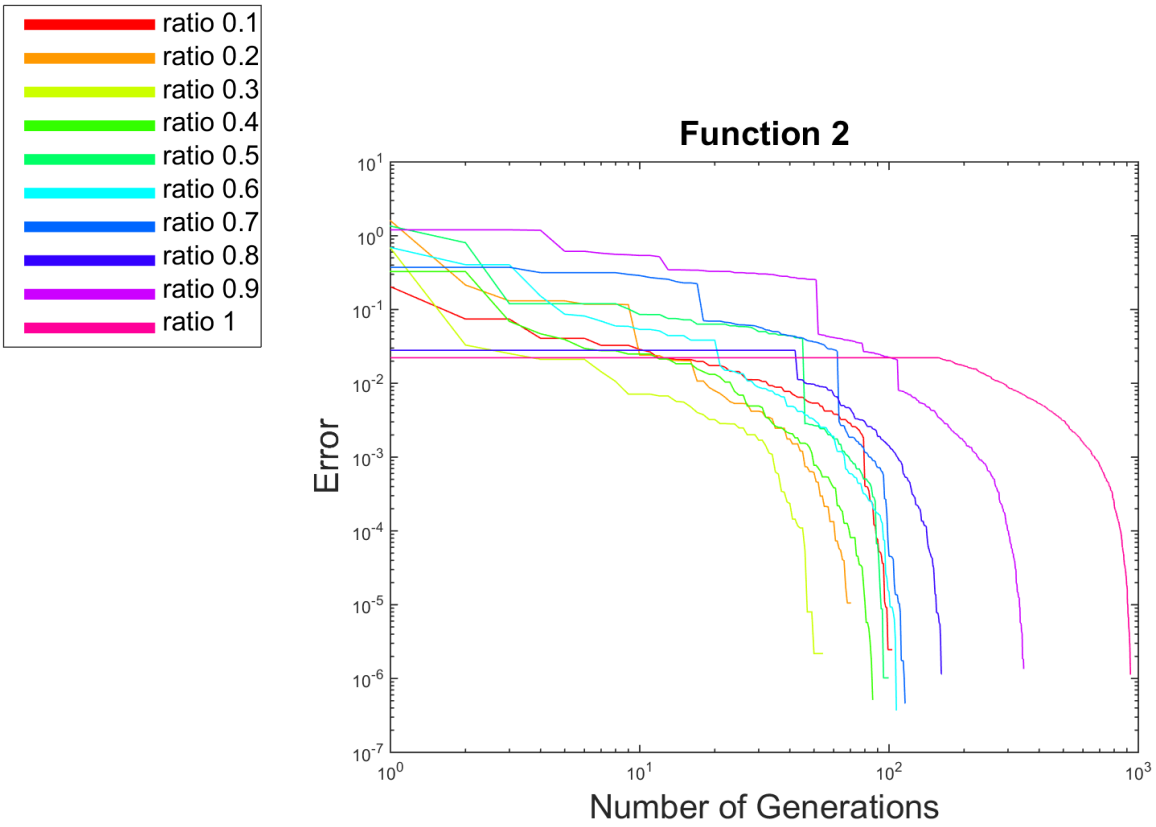


Figure 20.b

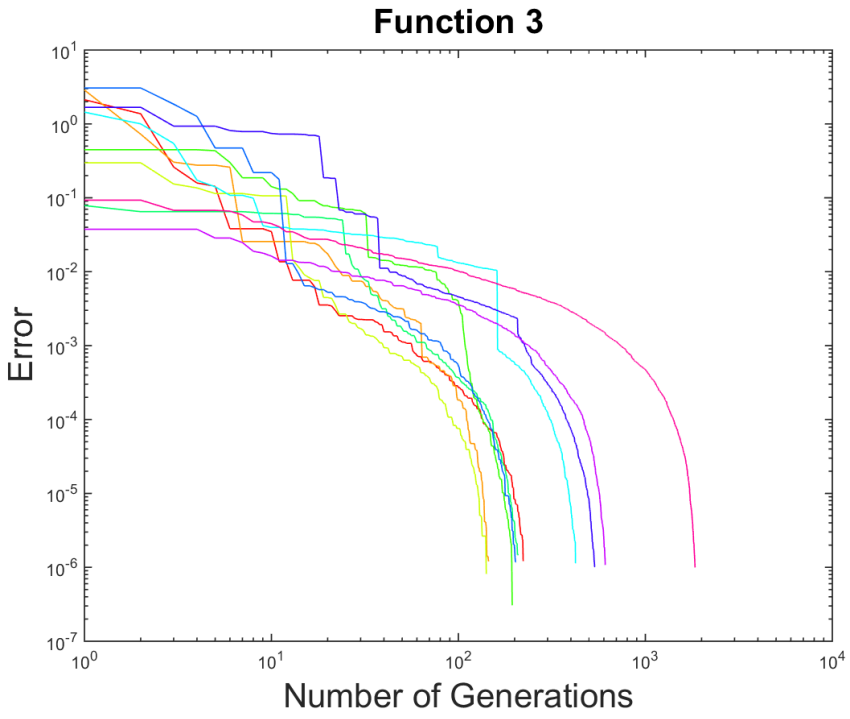


Figure 20.c

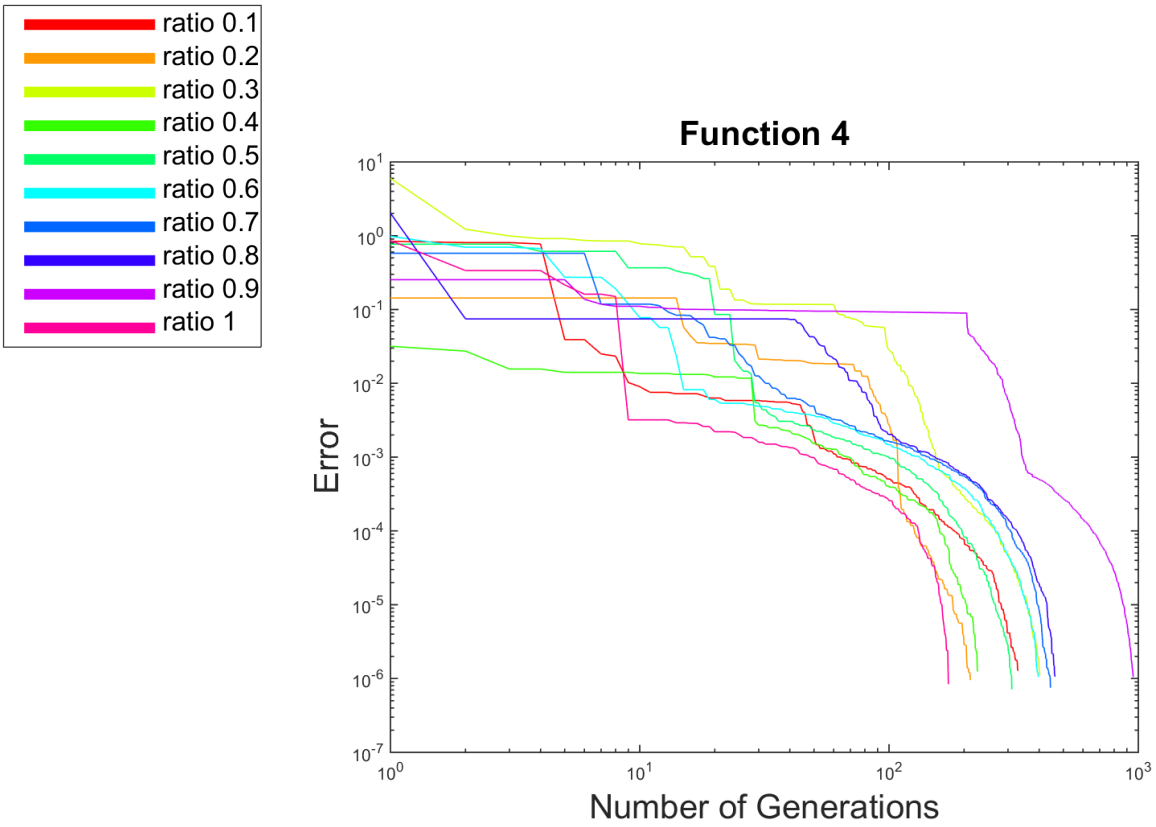


Figure 20.d

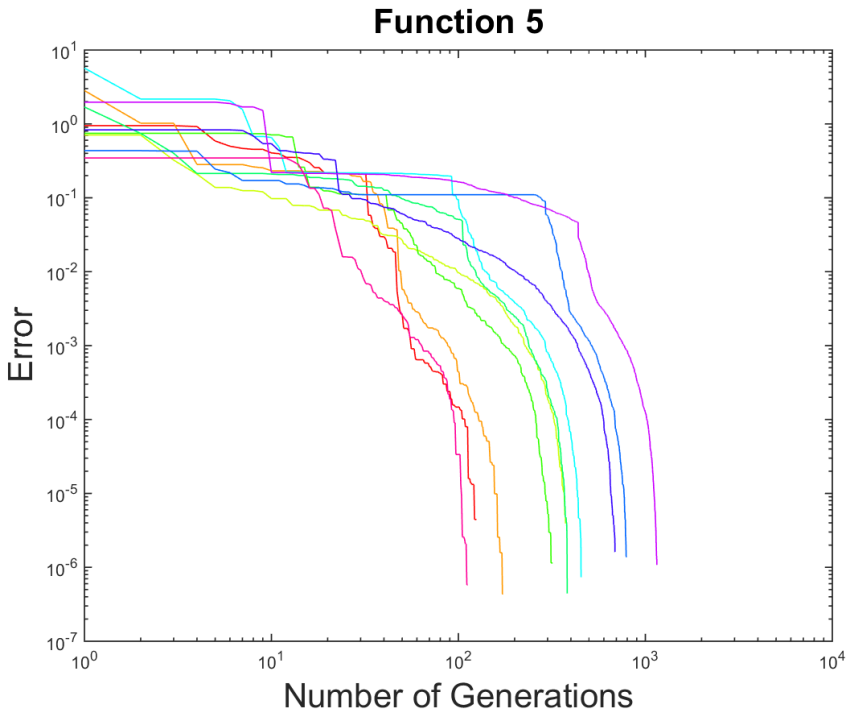


Figure 20.e

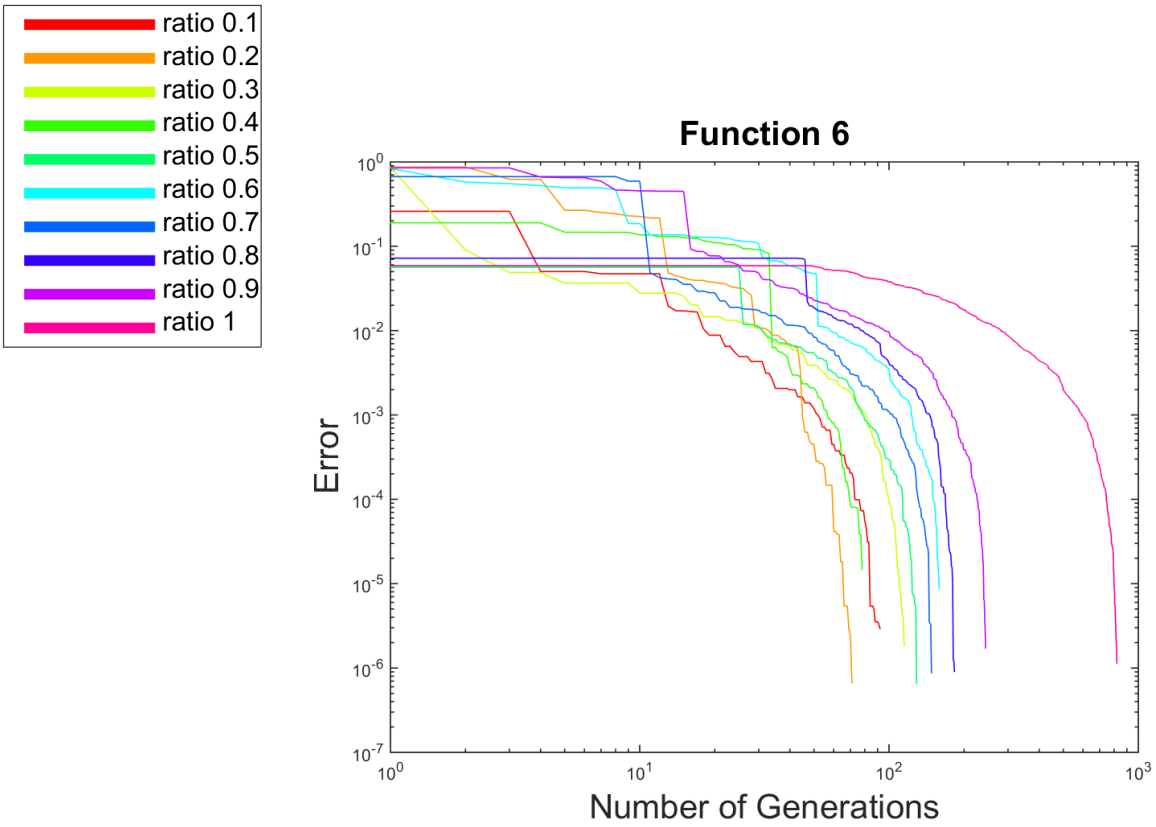


Figure 20.f

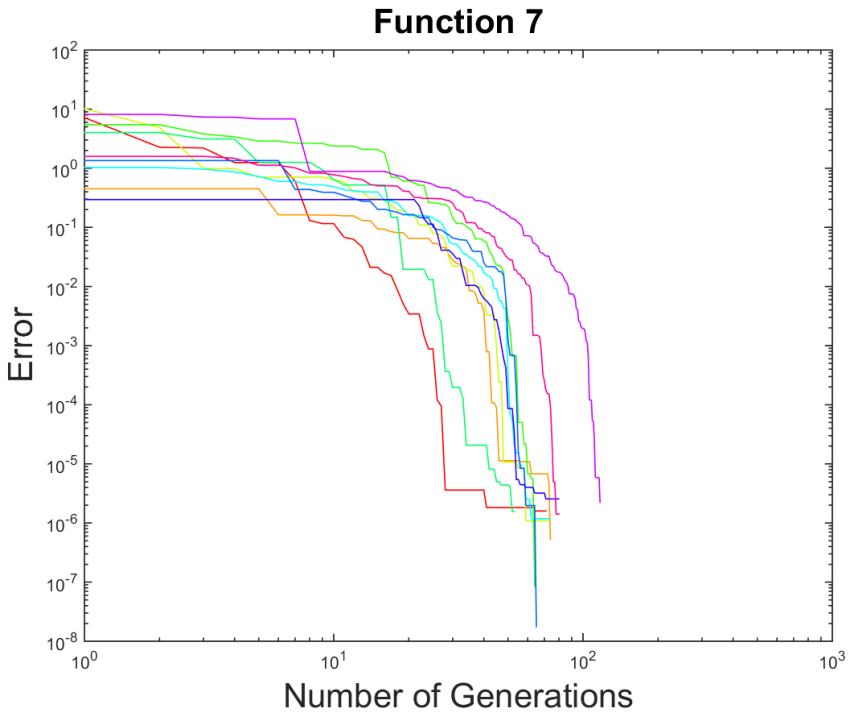


Figure 20.g

Figure 21: Spread in error for adjusting the ratio the size of xBest to the population size with mutation

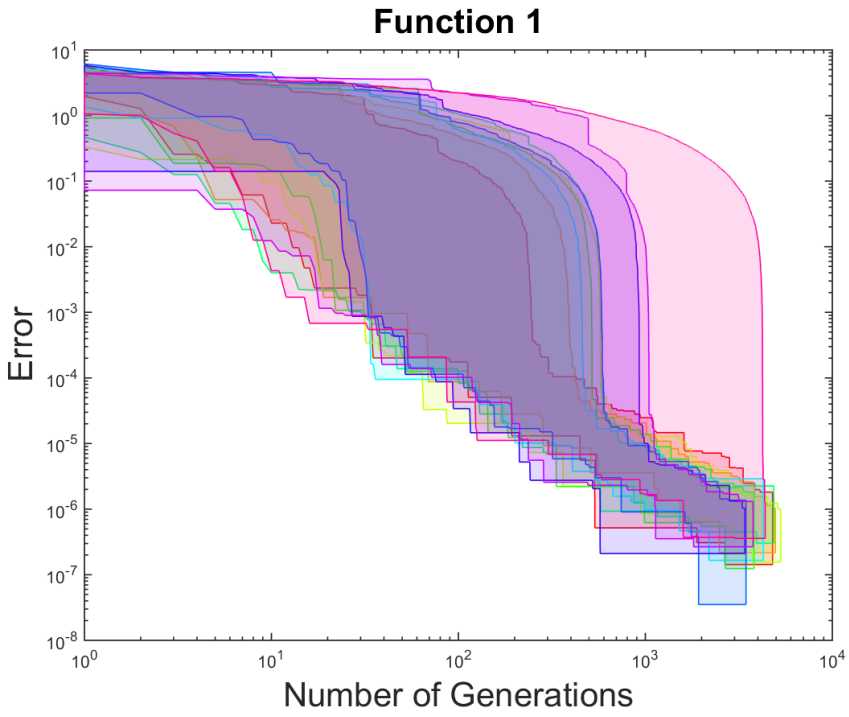
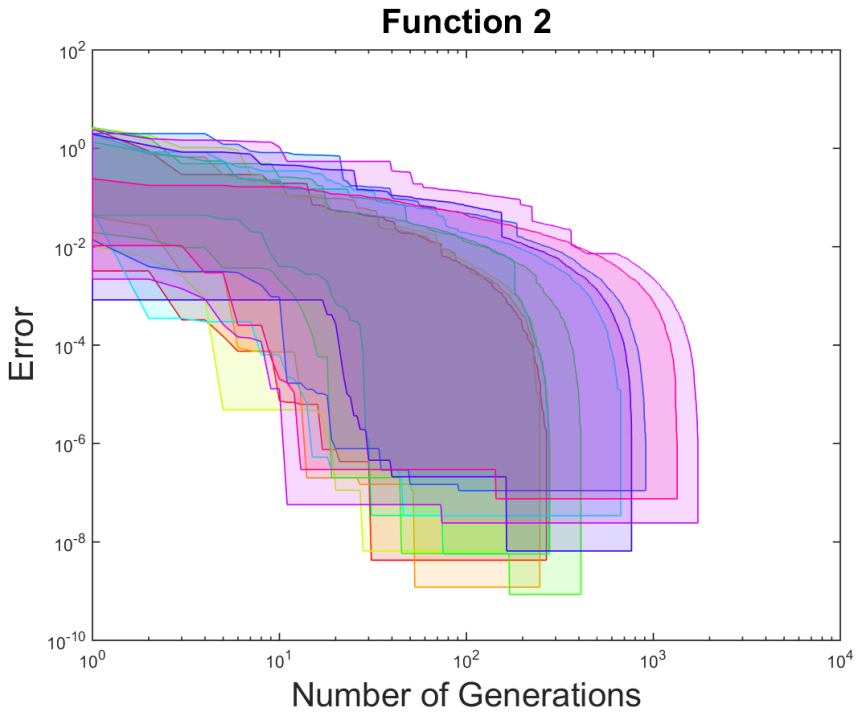


Figure 21.a



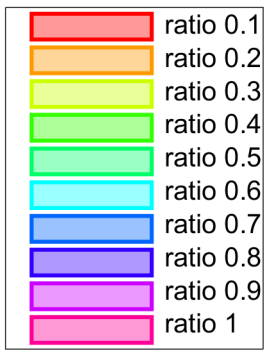


Figure 21.b

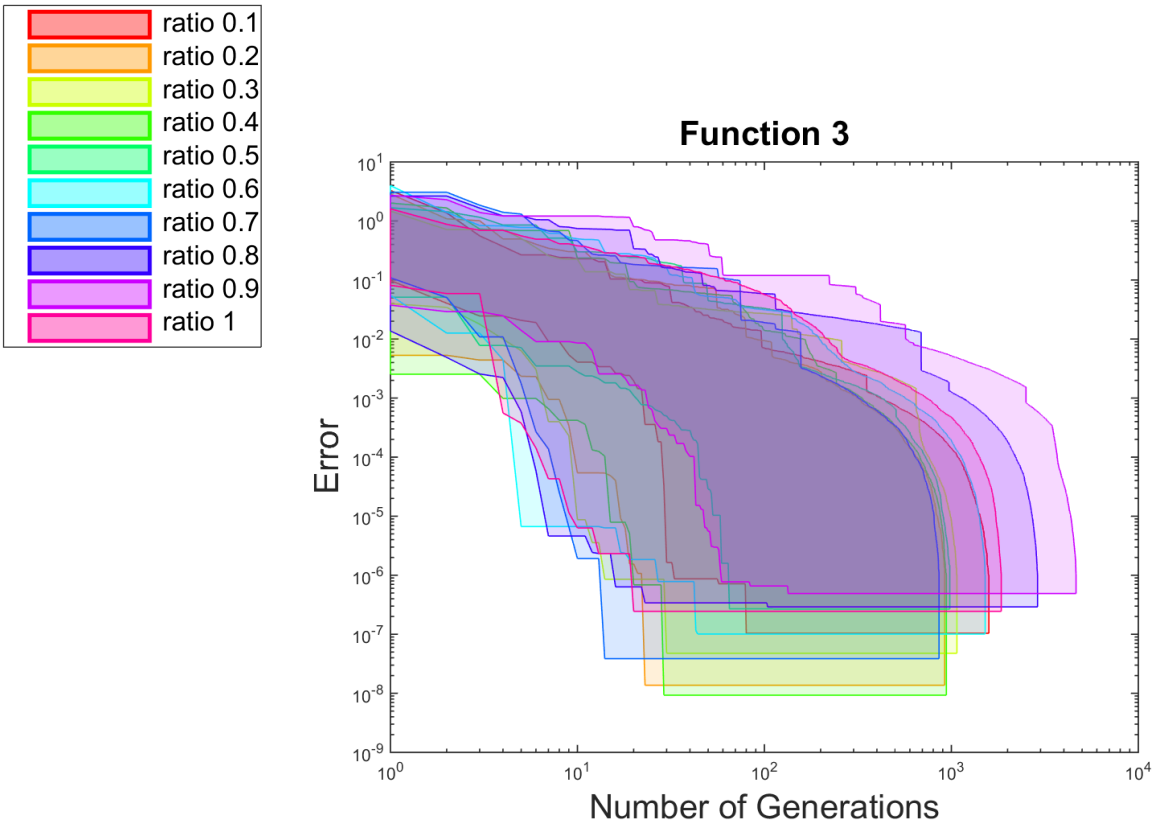


Figure 21.c

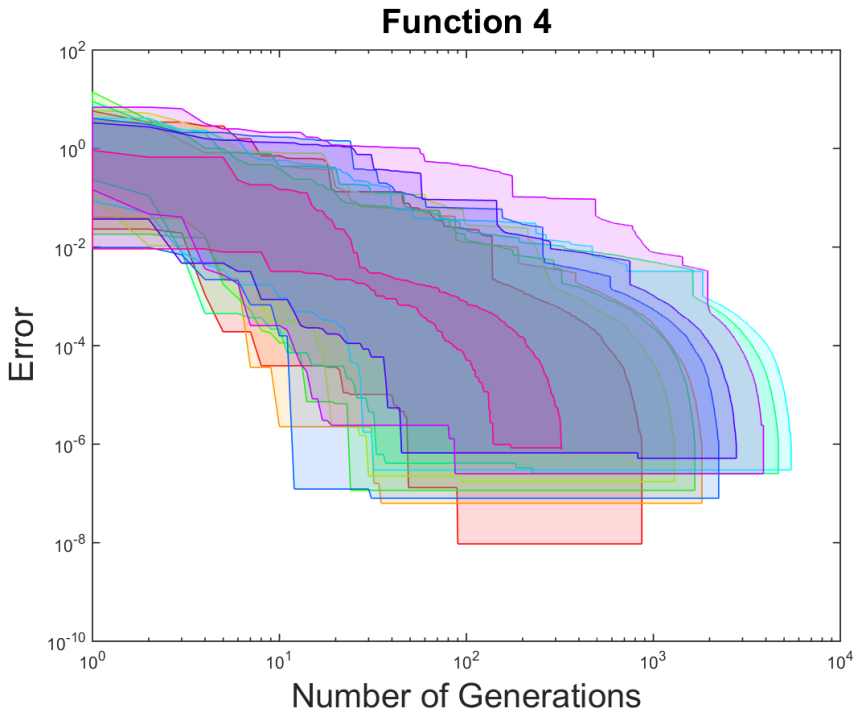


Figure 21.d

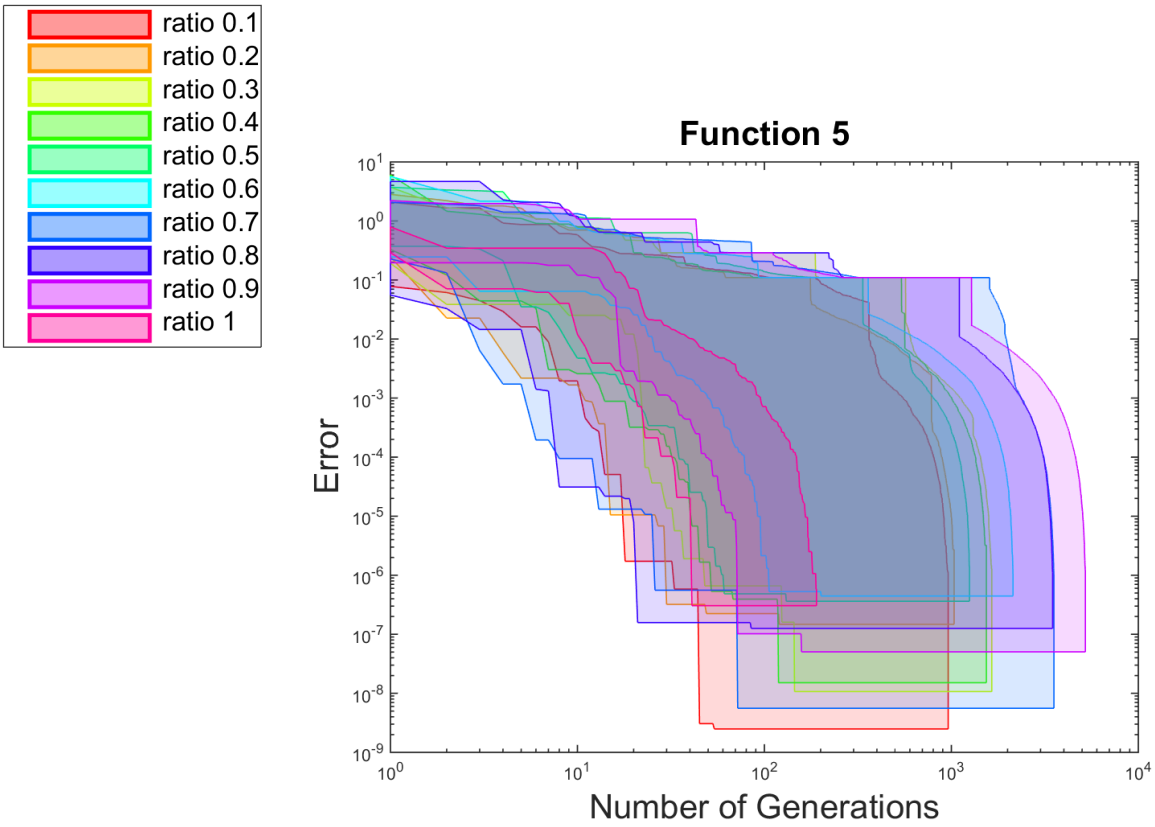


Figure 21.e

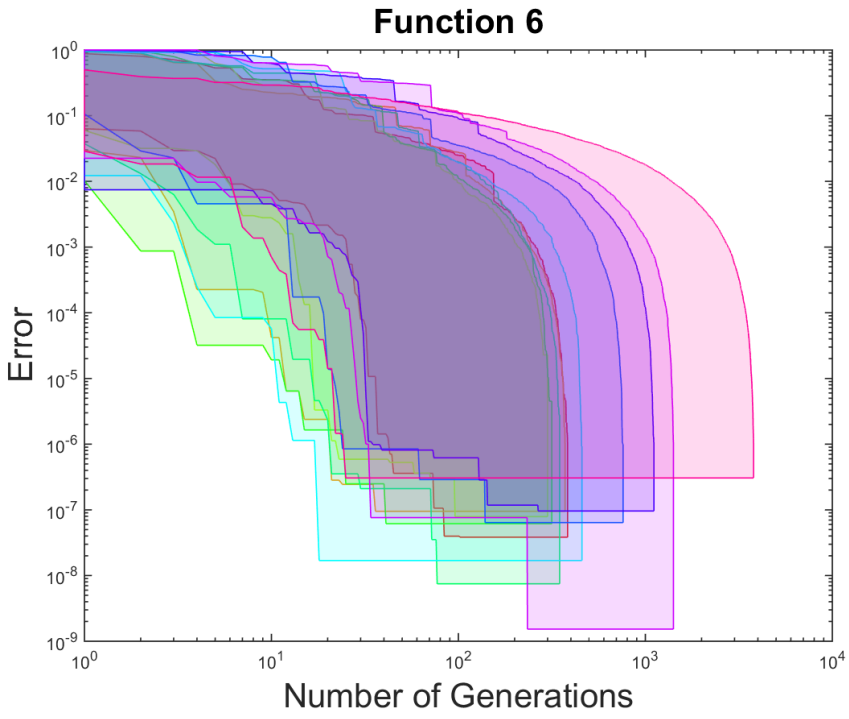


Figure 21.f

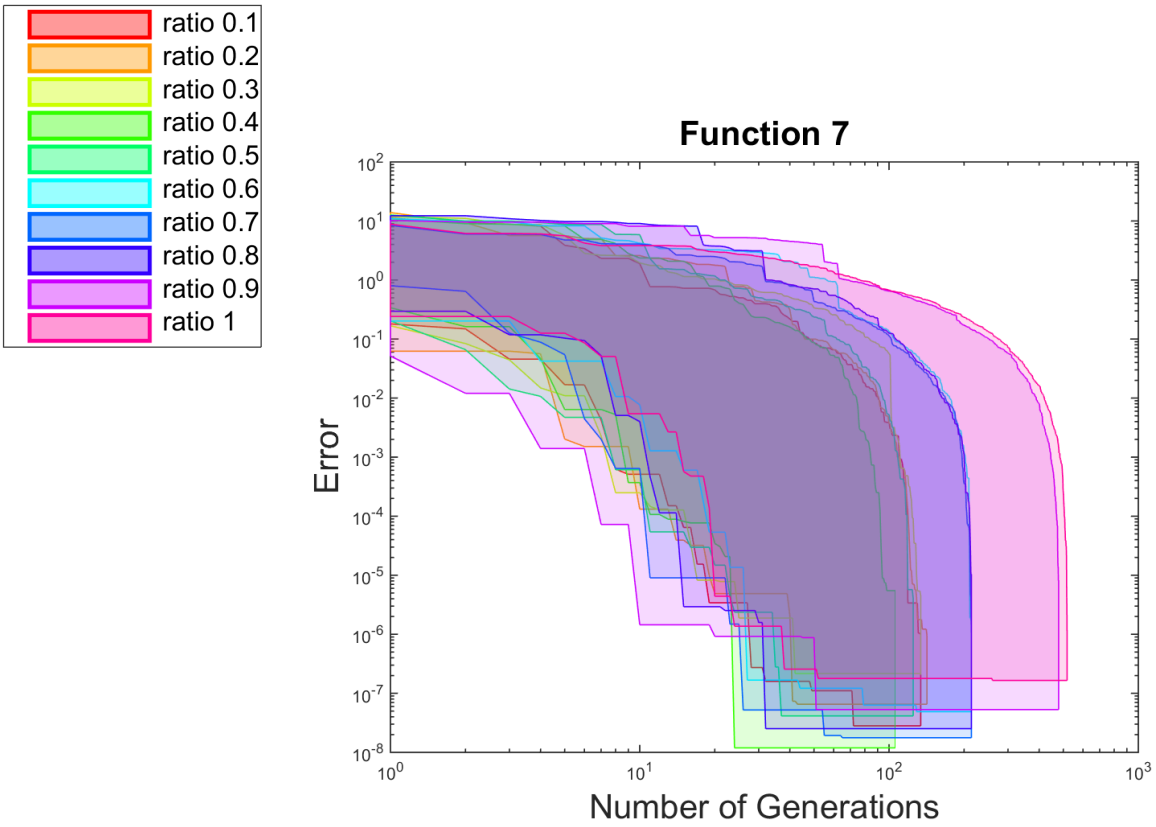


Figure 21.g

- The spread in Figures 21 are large and vary a lot. This means that the error decays from the different ratios are similar and do not make too much of a difference.
- However, the ratio of 1 seems to have the slowest error decay for most functions as seen from Figures 20.
- In terms of error decay, the conclusion we get is that ratio of 1 gives the slowest error decay and the lower ratios give the faster error decays for mutation.

3.6 Number of Elements Needed to Evaluate Before Starting a New Generation - Crossover

Table 7: Parameters used to get the data

Ratio of xBest to the population size	0.6
Ratio of number of elements to evaluate before starting a new generation to the population size	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Number of workers	5
Repeats	20
Population size	50
Priority to the latest generation	0.7
Maximum Time	60 seconds
Tolerance	1×10^{-6}
Functions used	1, 2, 3, 4, 5, 6, 7
Breeding type	Crossover

Figure 22: Results for adjusting the ratio of the number of elements needed to evaluate before starting a new generation to the population size with crossover

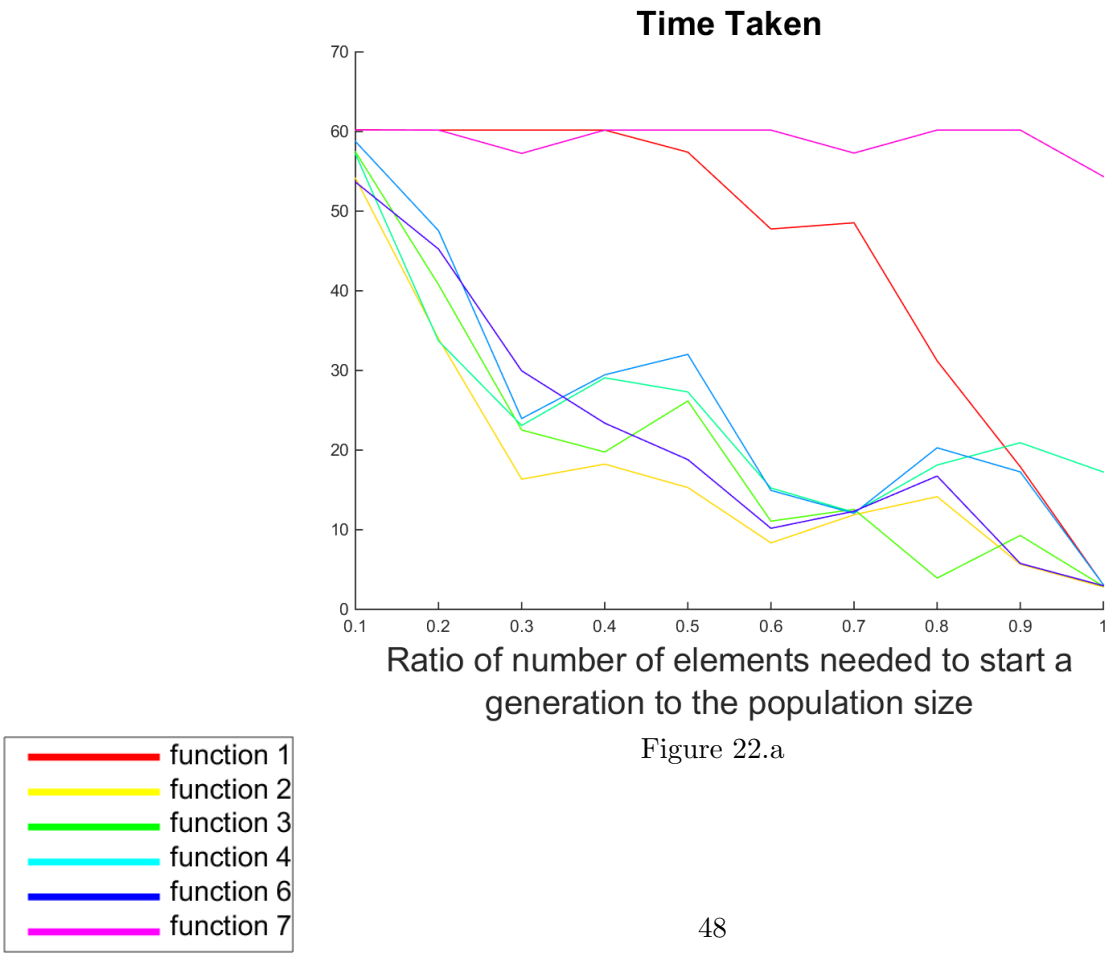


Figure 22.a

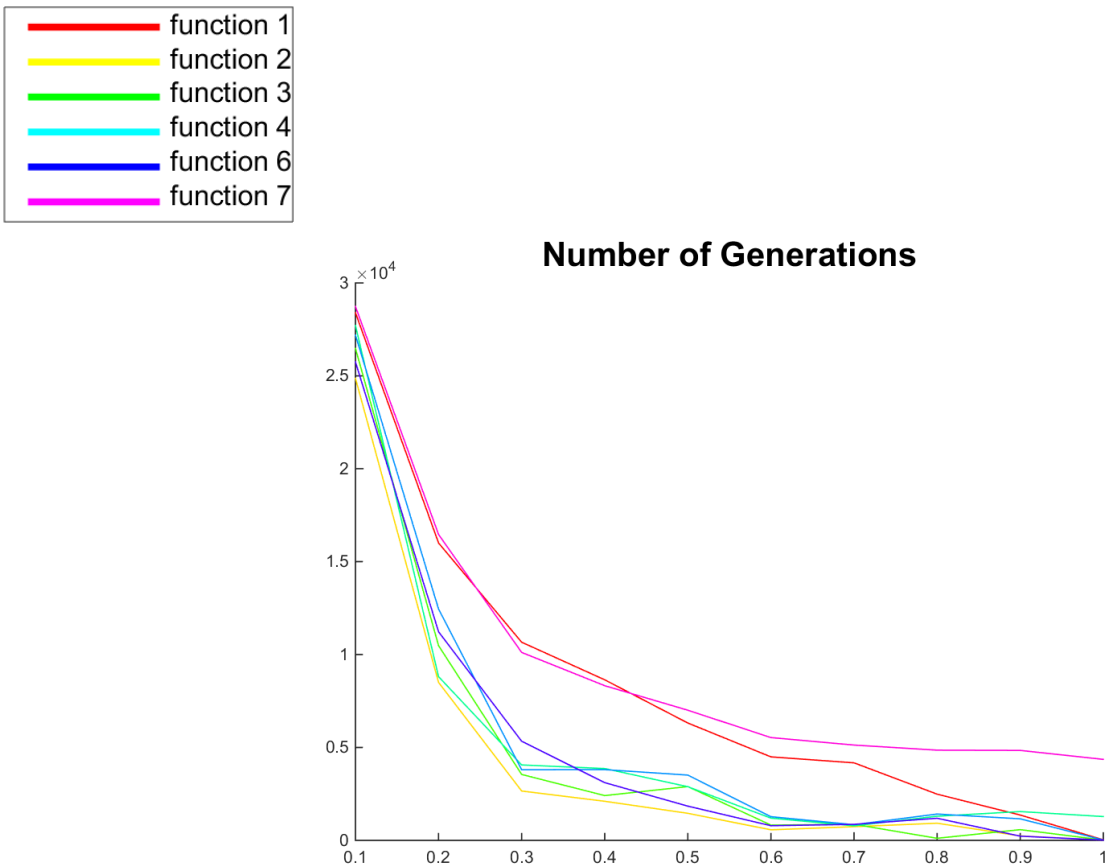


Figure 22.b

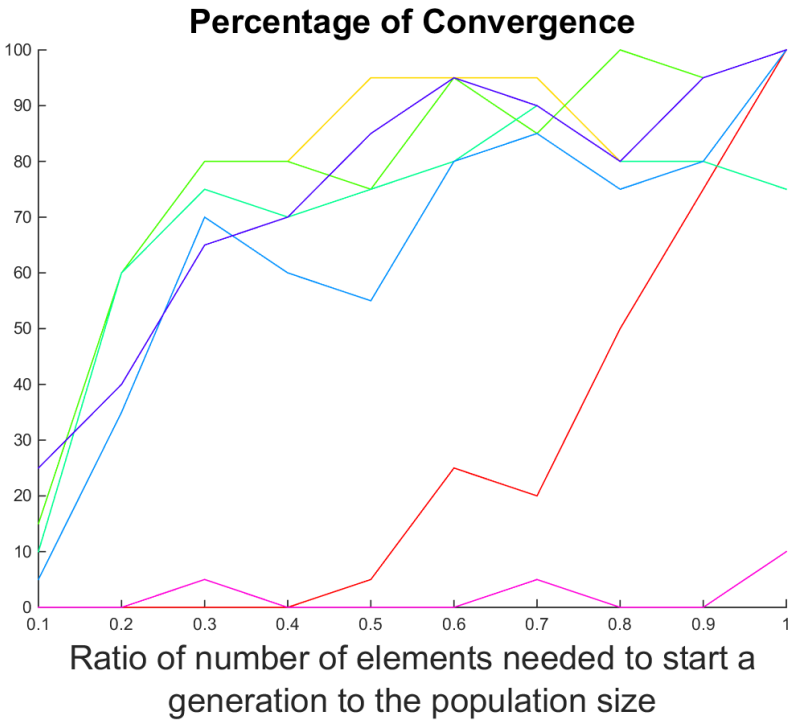


Figure 22.c

- Figure 22.c shows us that the percentage of convergence for most functions is hardly ever 100%. This shows that crossover is not as good as mutation. We see that the percentage of convergence increases with the ratio. This is because we get more information as the ratio increases, the more information we have, the better we can fit a Gaussian distribution to the data.
- We also see that function 7, hardly converges to the minimum. This is because this function have multiple global minima, this means that the Gaussian distribution does not describe the distribution of the points well. However, a Gaussian mixture model might work better for this function.
- From Figure 22.a, we see that when we do converge, we converge just as fast as mutation breeding and Figure 22.b shows that the number of generations reached is similar when we converge to the minimum. The number of generations reached decreases with the ratio as it takes longer to a new generation to be made as the higher the ratio, the more elements need to be evaluated.
- In conclusion, for crossover, we get the best results for ratio equal to 1.

Figure 23: Errors for adjusting the ratio of the number of elements needed to evaluate before starting a new generation to the population size with crossover

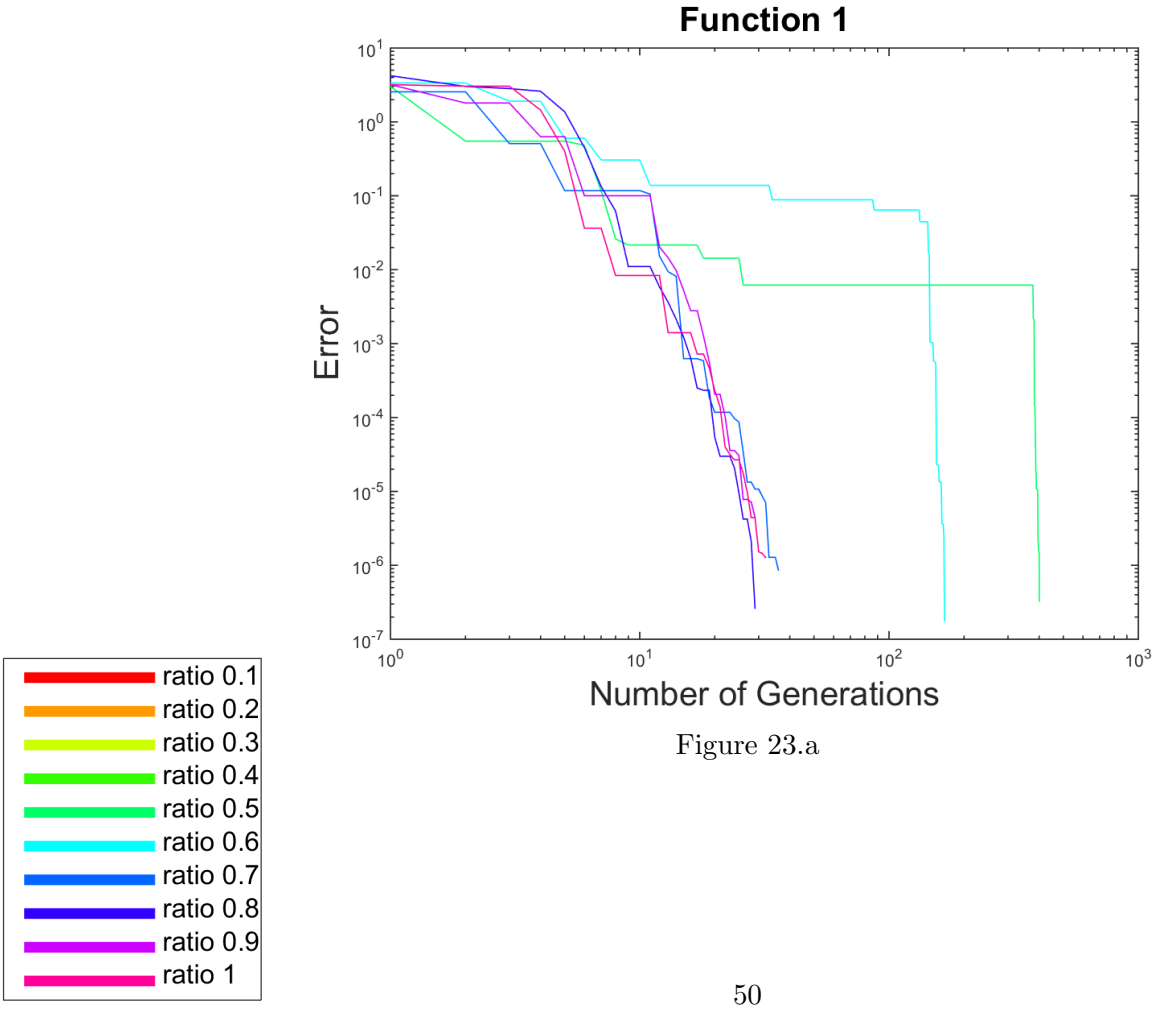


Figure 23.a

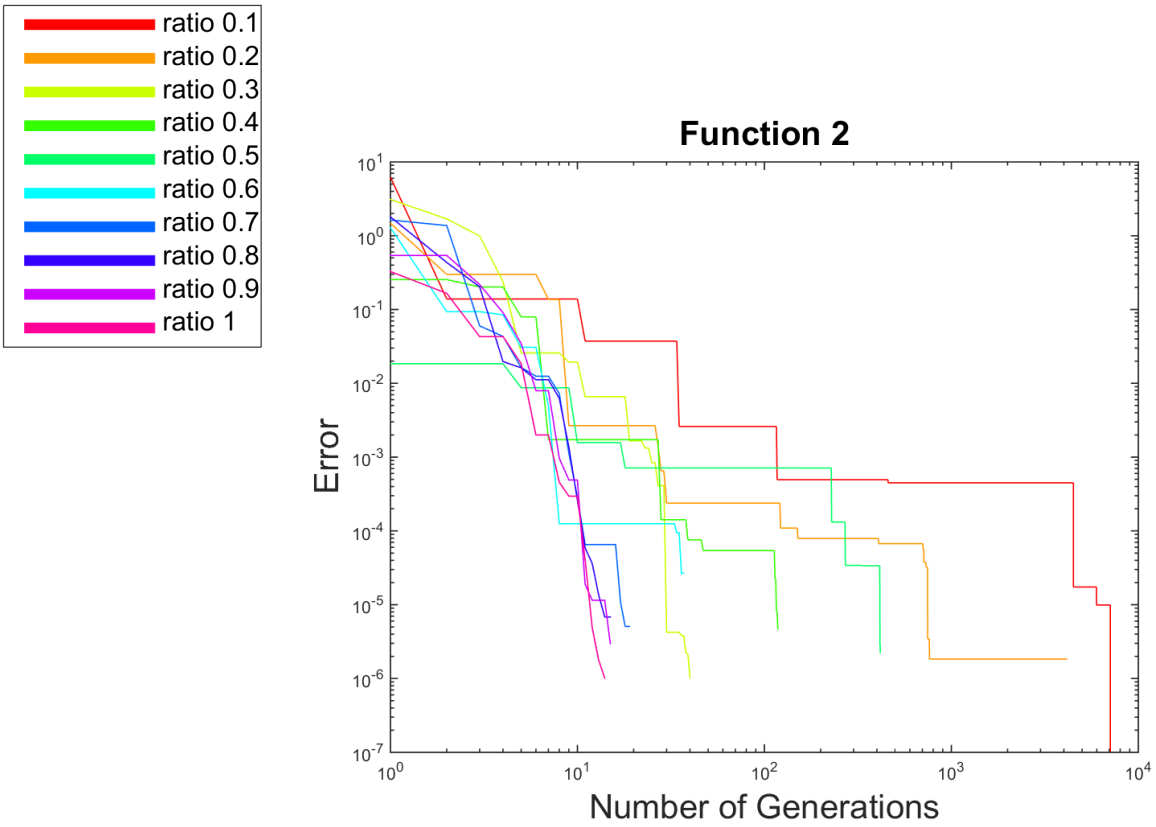


Figure 23.b

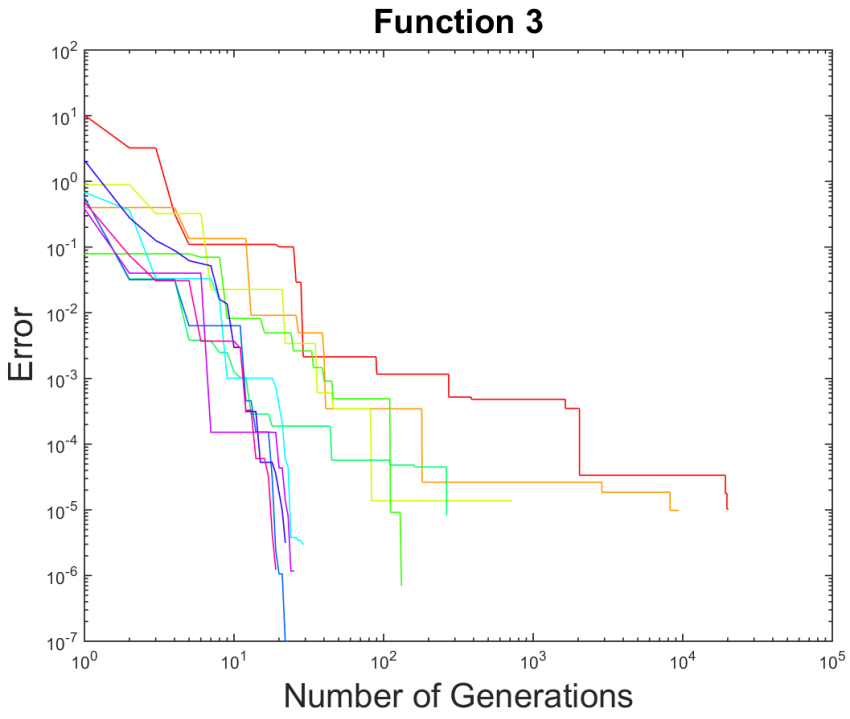


Figure 23.c

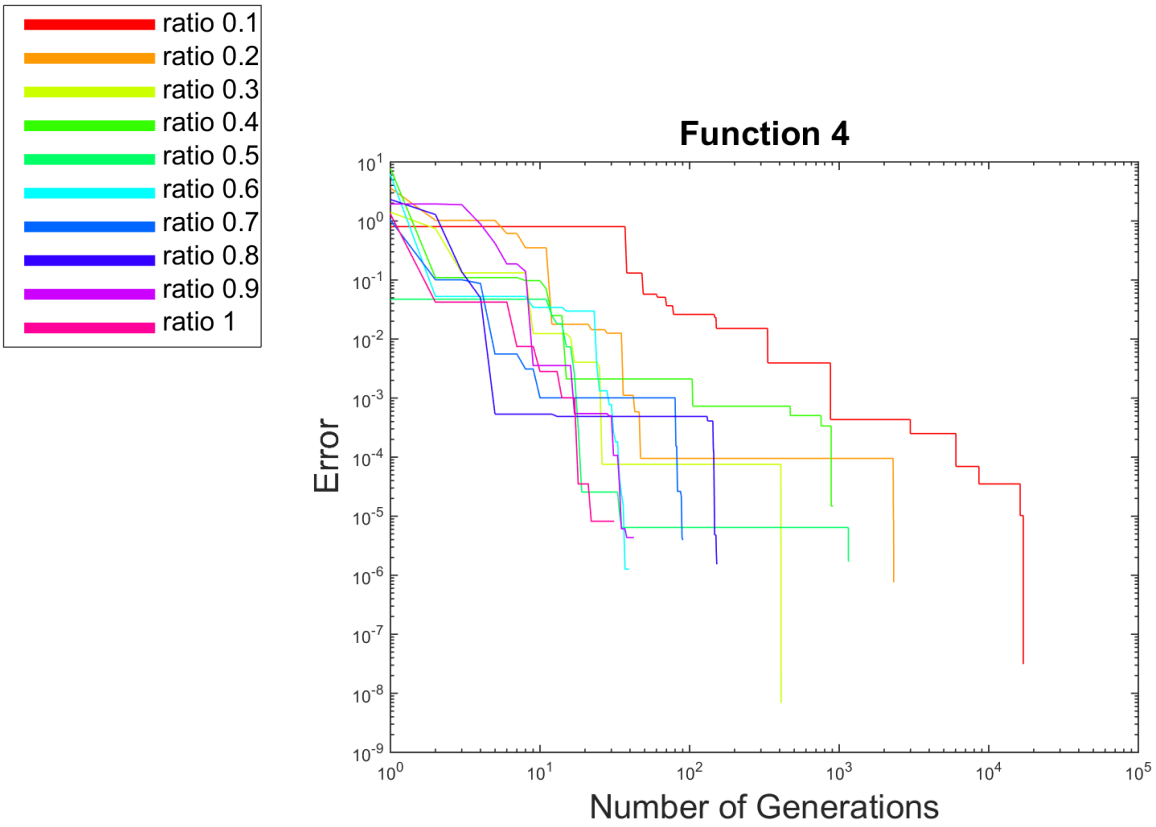


Figure 23.d

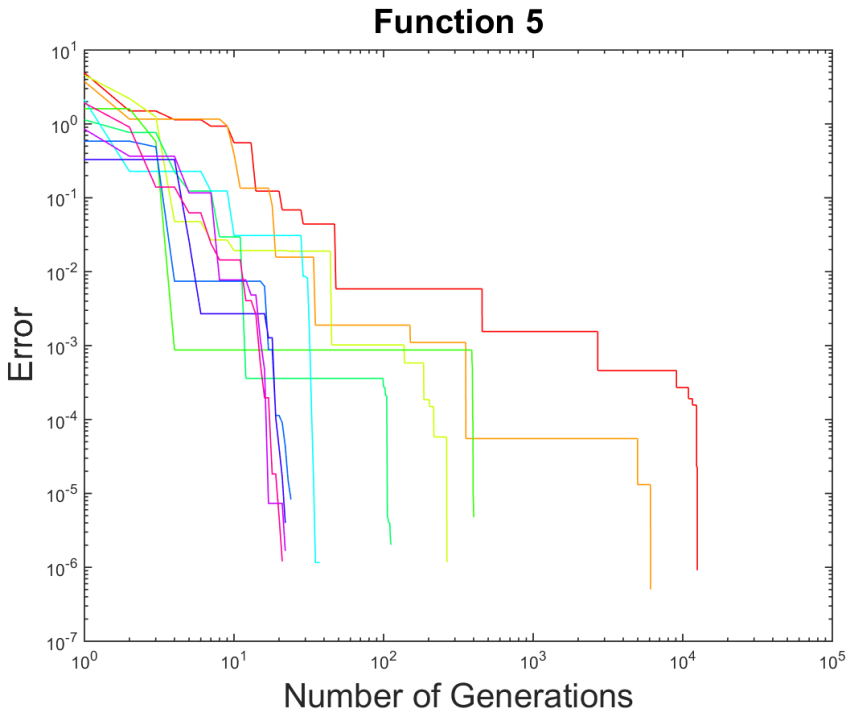


Figure 23.e

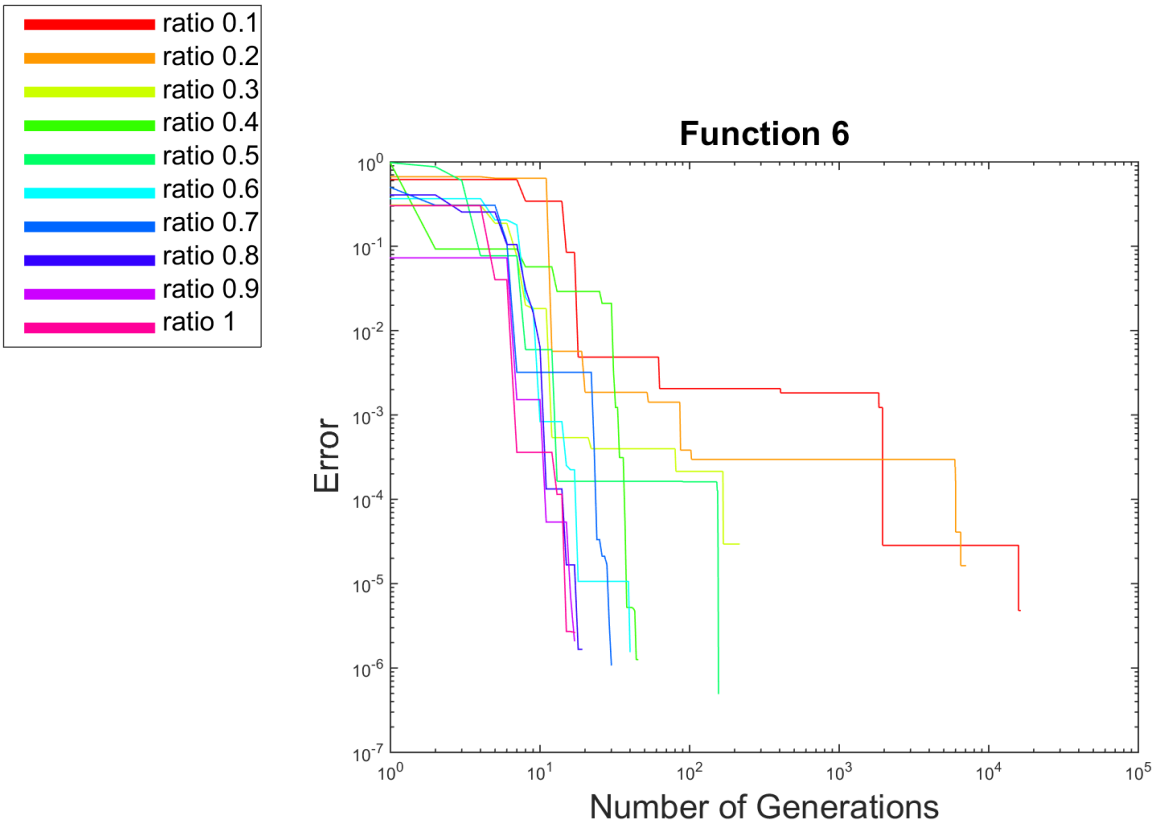


Figure 23.f

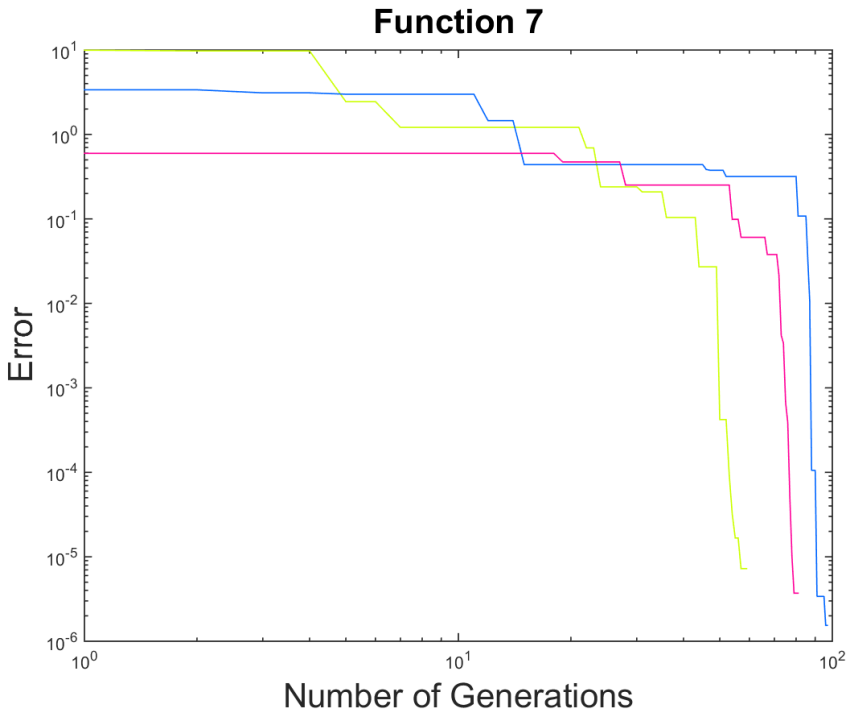


Figure 23.g

Figure 24: Spread in error for adjusting the ratio of the number of elements needed to evaluate before starting a new generation to the population size with crossover

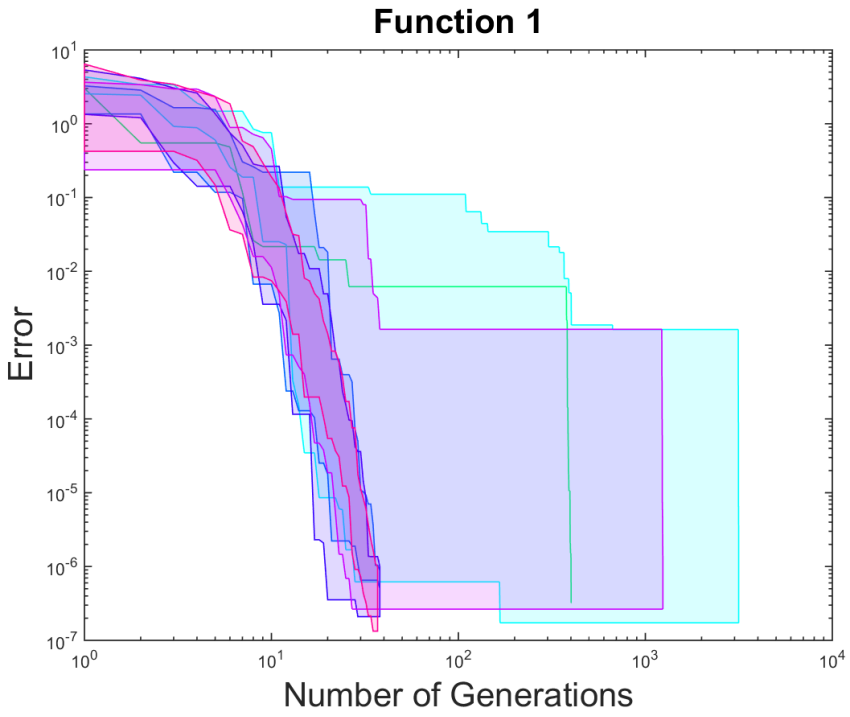


Figure 24.a

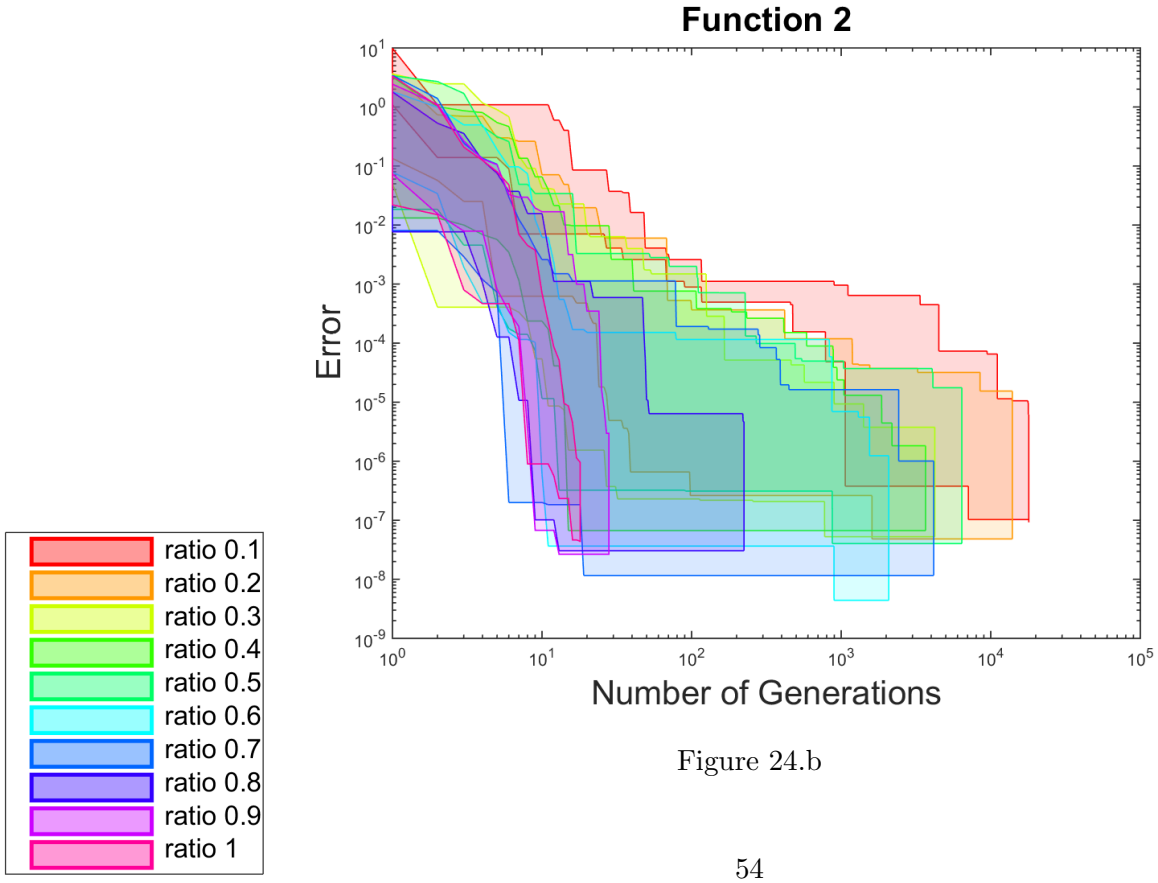


Figure 24.b

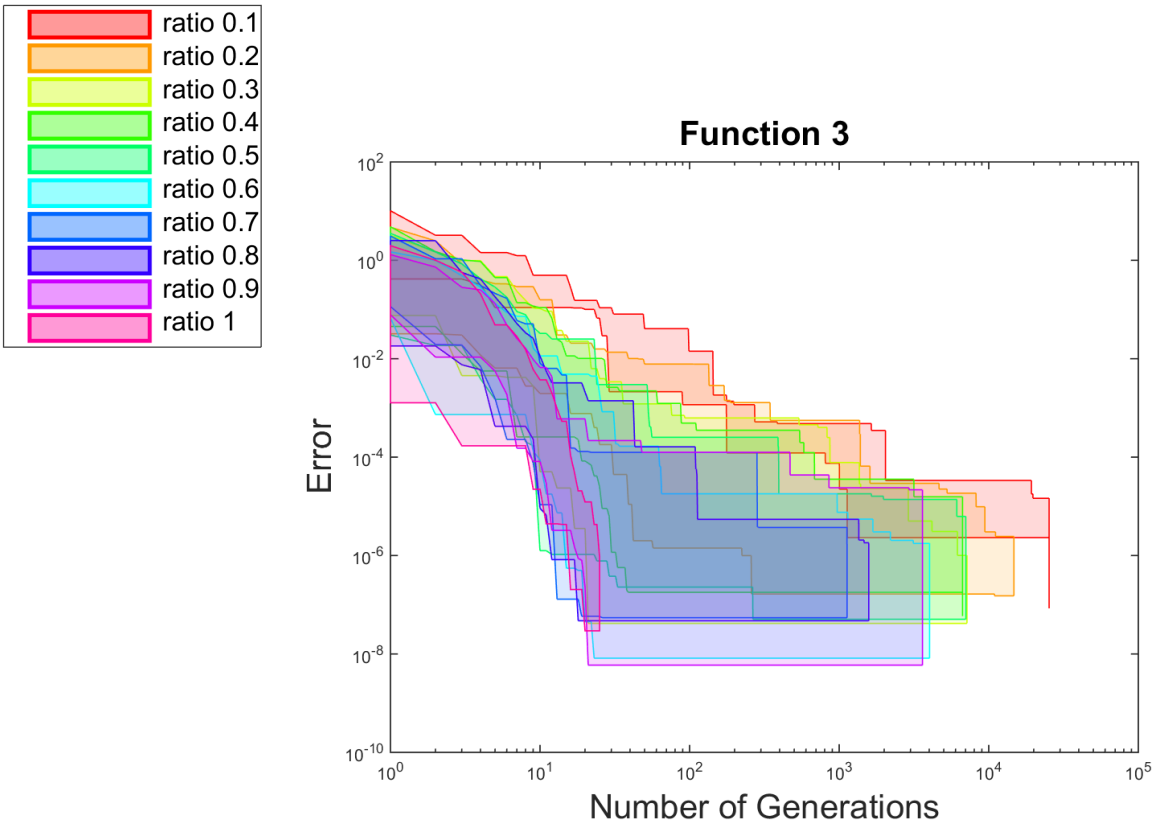


Figure 24.c

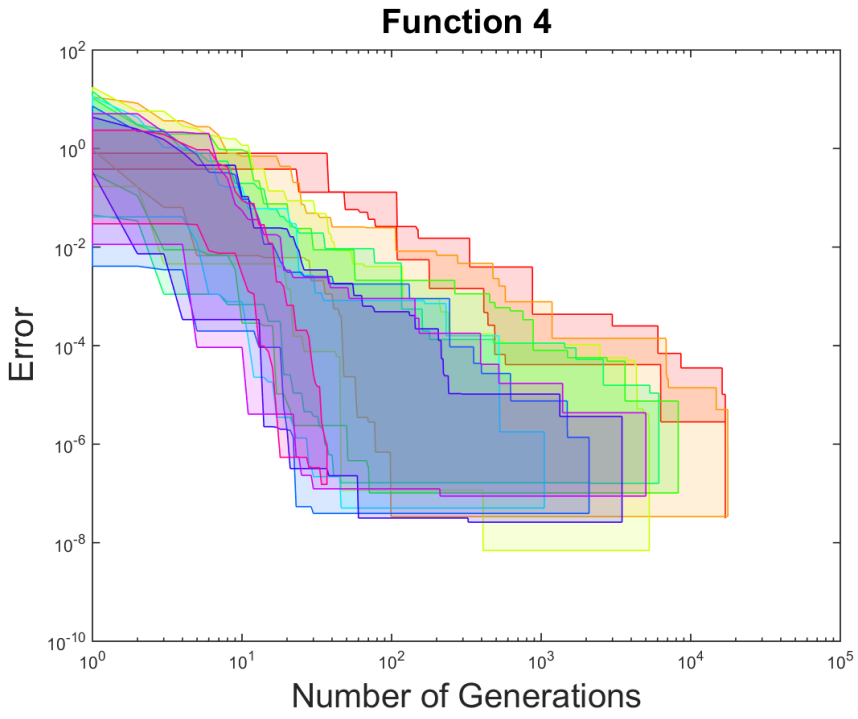


Figure 24.d

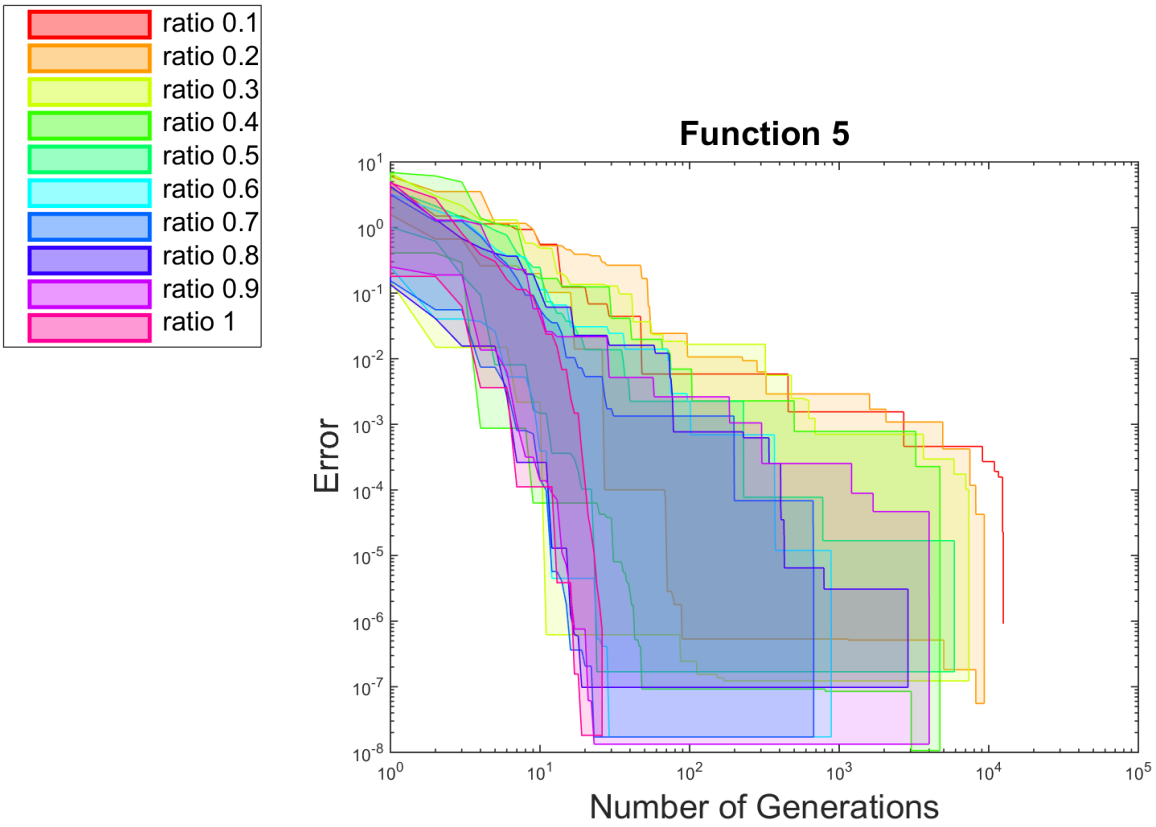


Figure 24.e

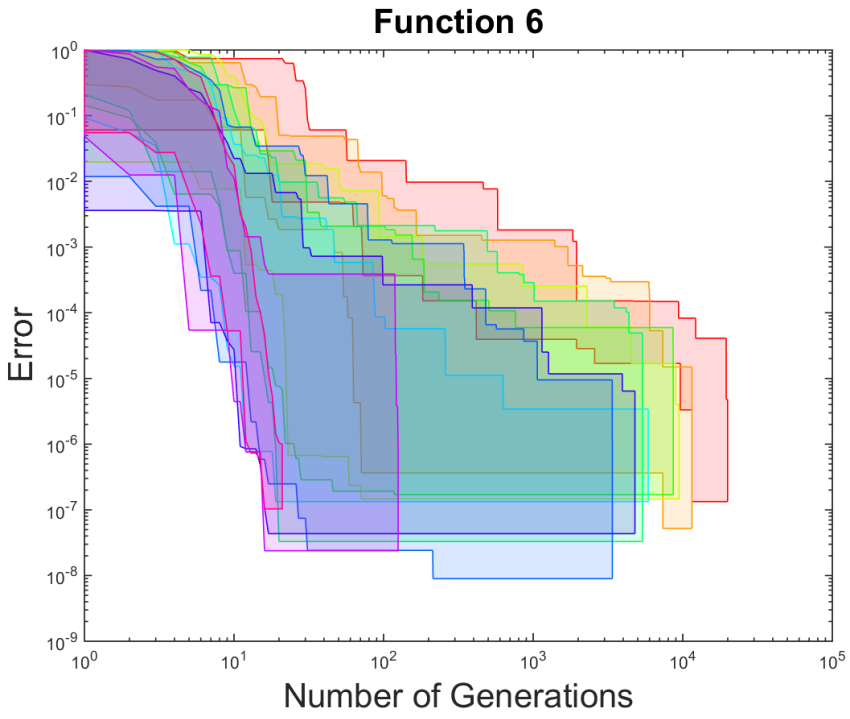


Figure 24.f

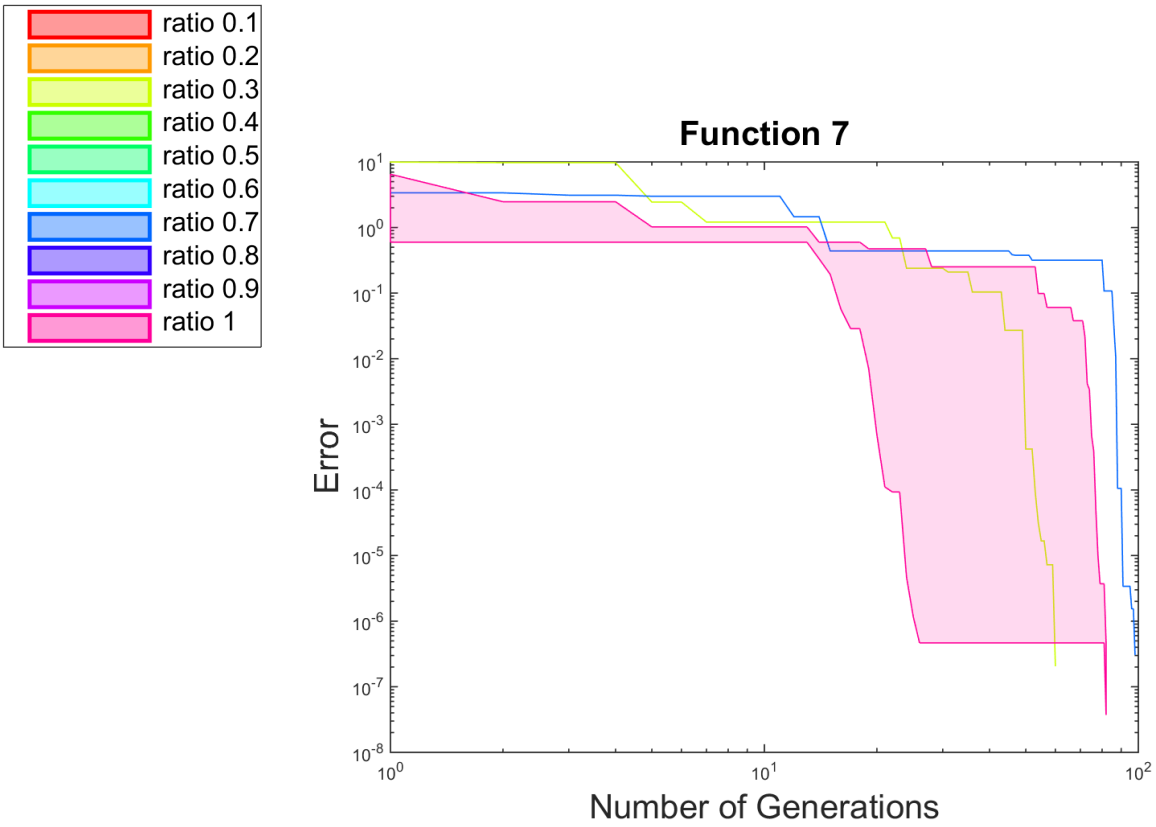


Figure 24.g

- From Figures 23, for functions 1-6, the ratios between 0.8 and 1 have the fastest error decays and where convergence to the minimum is reached, ratio of 0.1 and 0.2 have the slowest error decay. This shows that the higher the ratio, the better error decay we get.
- From Figures 24, we see that the spread is small for ratios of 0.1 and 1 for most functions, however the spread is much larger for ratios in the middle, especially 0.7-0.9. However, the picture is still clear with the error decay is faster as the ratio increases and is the best for ratio of 1, implying that this breeding works the best when using all of information available.
- In conclusion, for crossover, we get the best error decay for ratio equal to 1.

3.7 Size of xBest - Crossover

Table 8: Parameters used to get the data

Ratio of xBest to the population size	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Ratio of number of elements to evaluate before starting a new generation to the population size	0.6
Number of workers	5
Repeats	20
Population size	50
Priority to the latest generation	0.7
Maximum Time	60 seconds
Tolerance	1×10^{-6}
Functions used	1, 2, 3, 4, 5, 6, 7
Breeding type	Crossover

Figure 25: Results for adjusting the ratio of the size of xBest to the population size with crossover

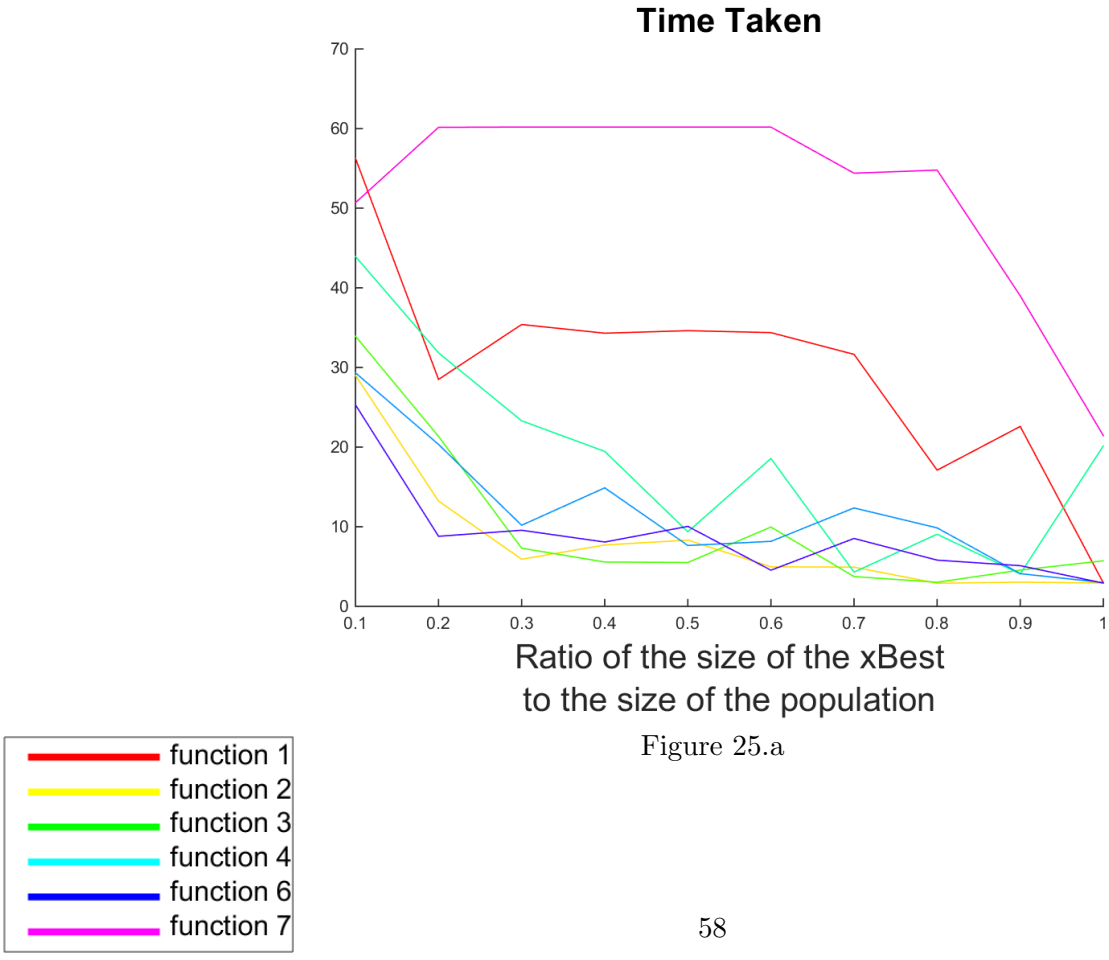


Figure 25.a

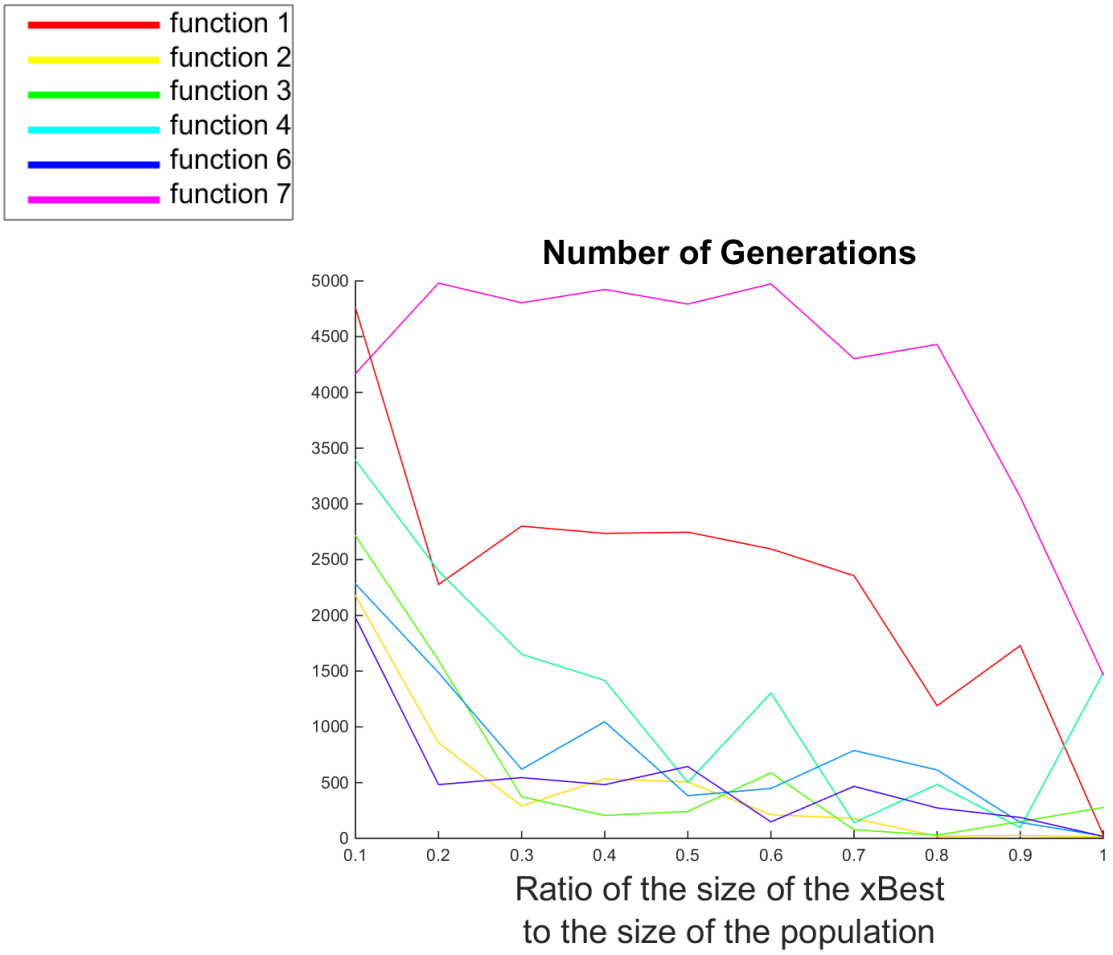


Figure 25.b

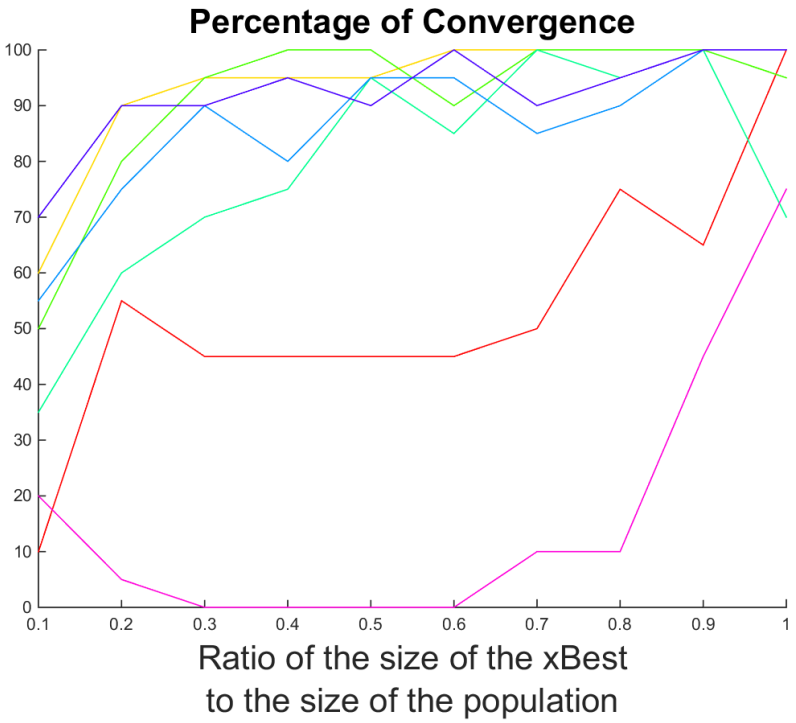


Figure 25.c

- From Figure 25.c, we can see that for ratio of 0.1, we get a low percentage of convergence to the minimum for functions 1-5 then increases between 0.1-0.4 and then stays flat. We get similar results for functions 6 and 7 but much lower. The most percentage of convergence is seen at ratio of 1. This is because we have the most number of elements to fit a distribution to and this gives us the best distribution to sample from for the next population.
- From Figures 25.a and 25.b, we see that the time taken and the number of generations reached are proportional. This is because the number of elements needed to evaluated for each generation is fixed. We see that the general trend for time taken is that it decreases as the ratio increases.
- In conclusion, for crossover, we get the best results for ratio equal to 1.

Figure 26: Errors for adjusting the ratio of the size of xBest to the population size with crossover

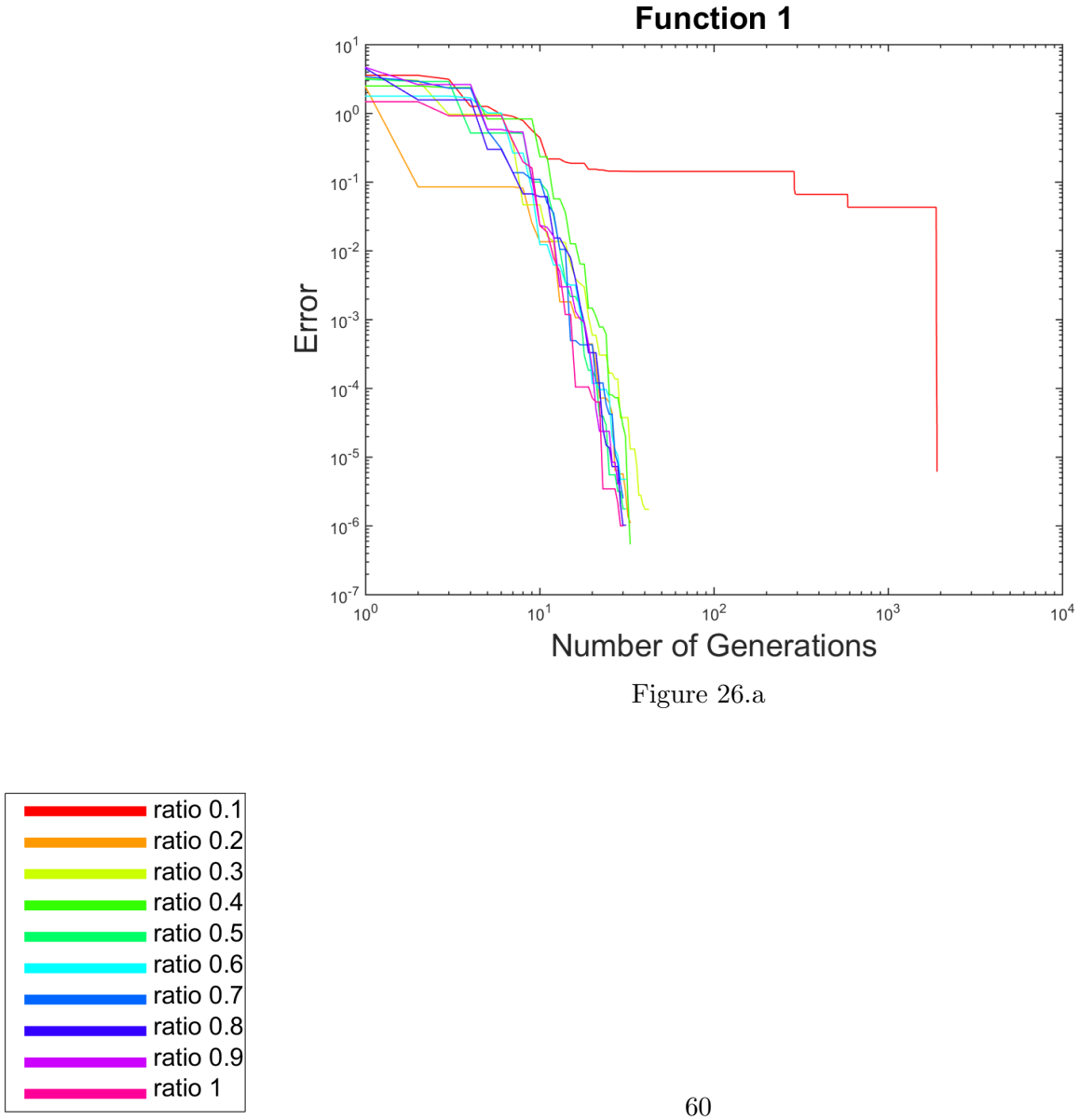


Figure 26.a

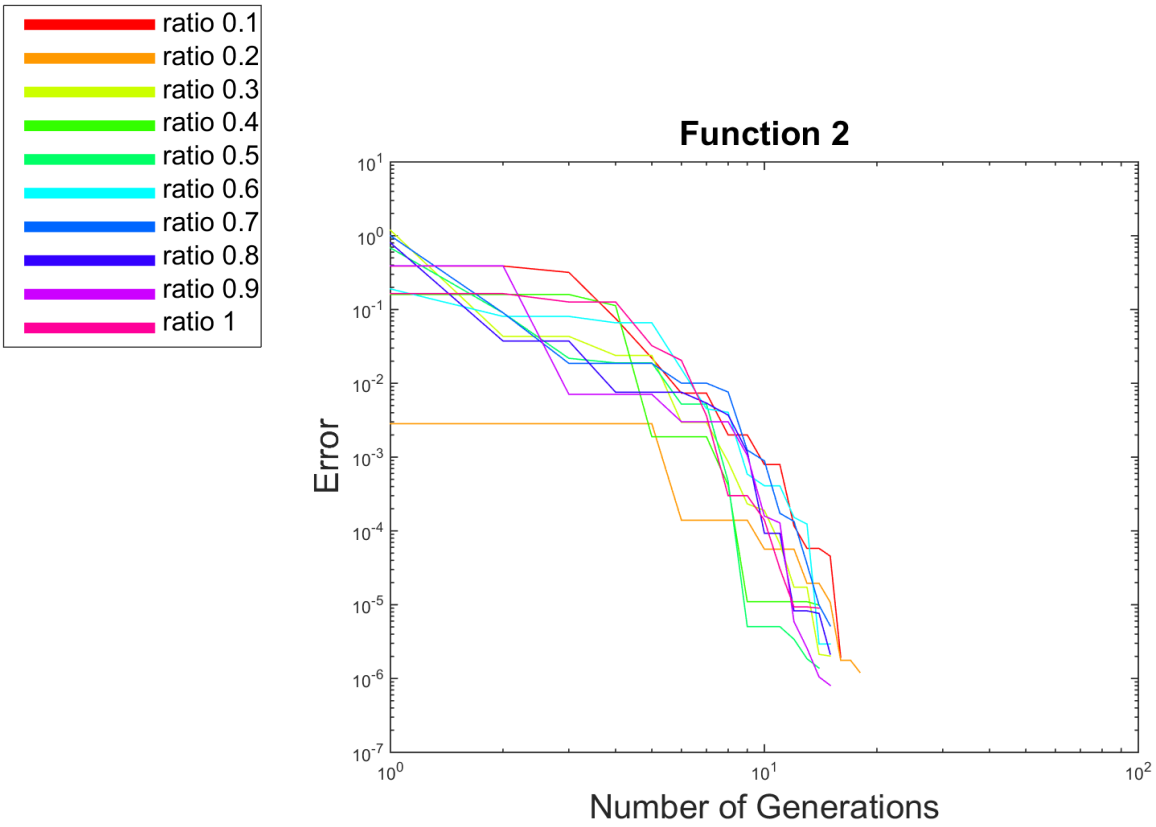


Figure 26.b

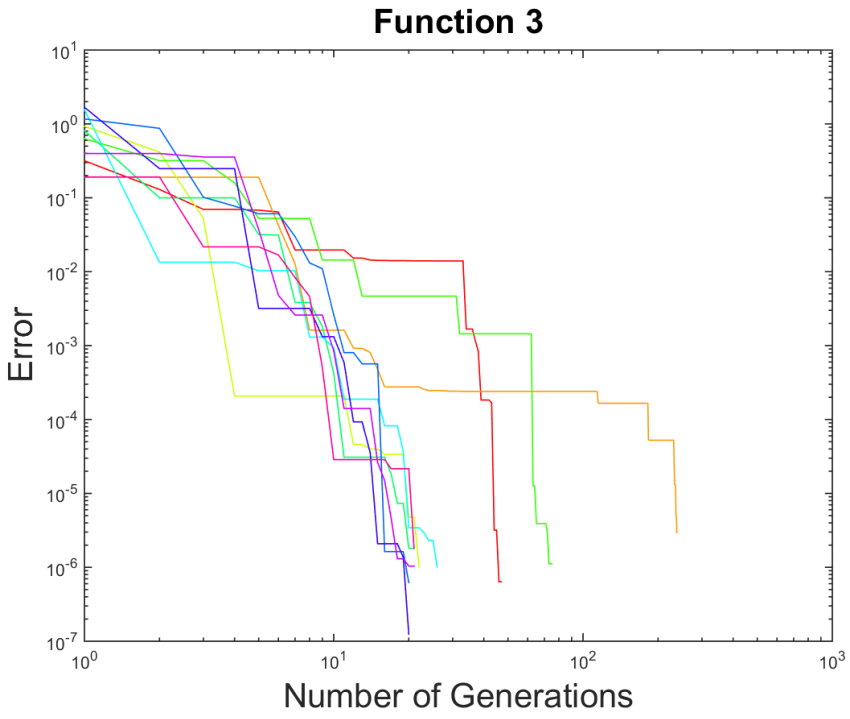


Figure 26.c

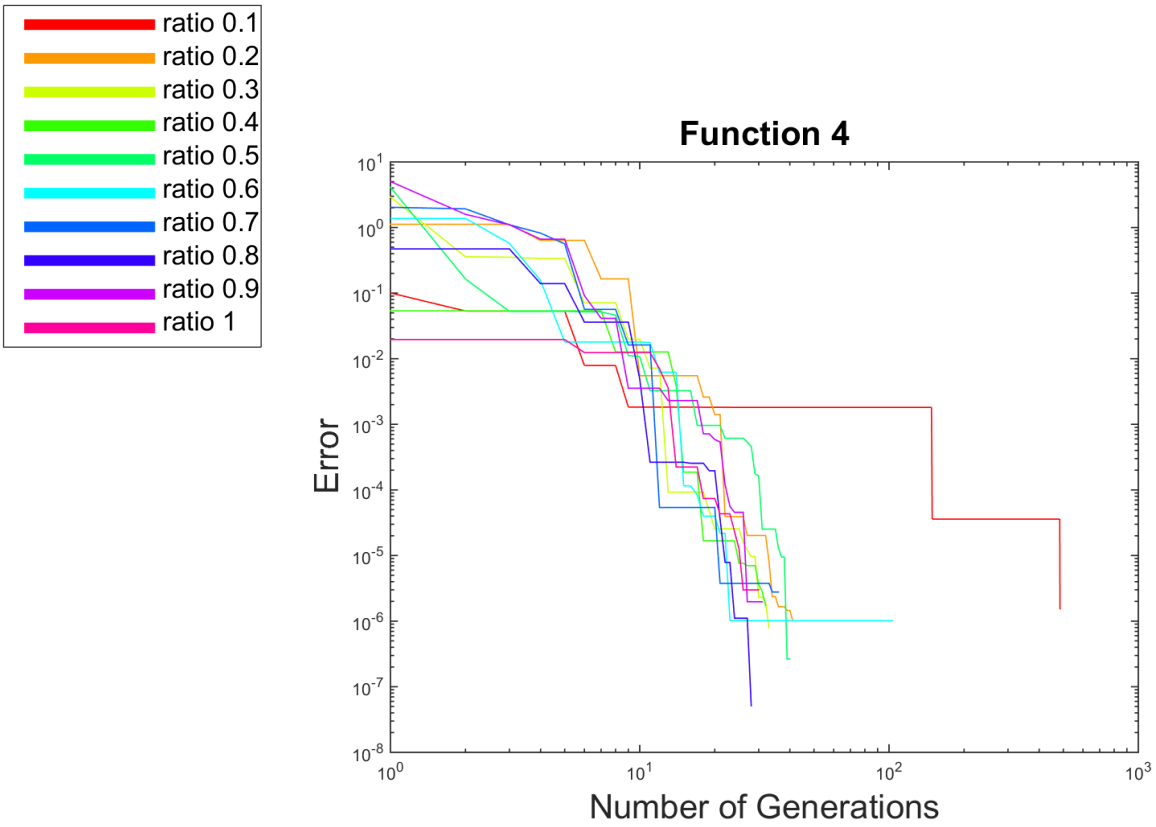


Figure 26.d

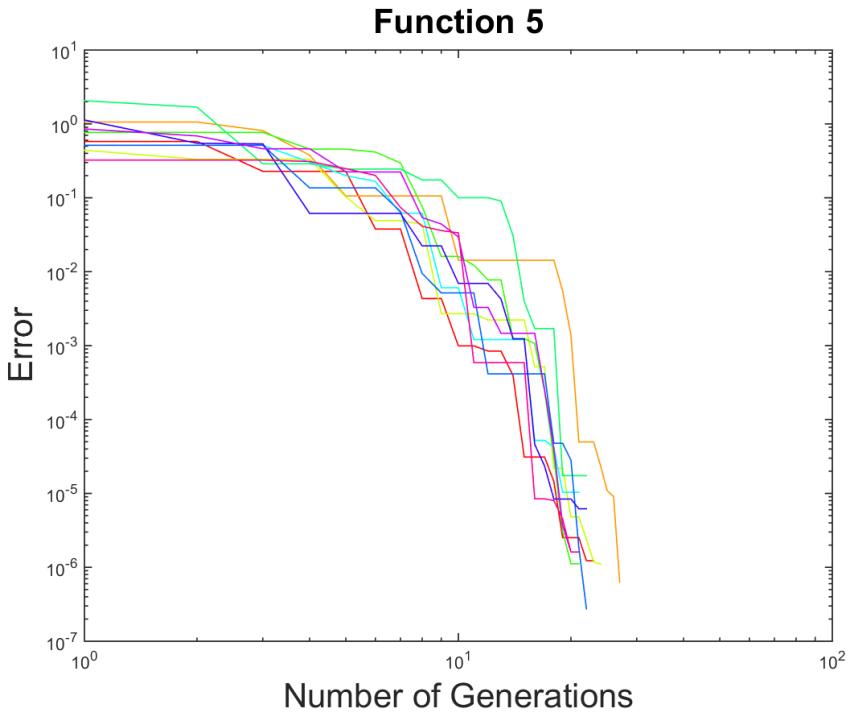


Figure 26.e

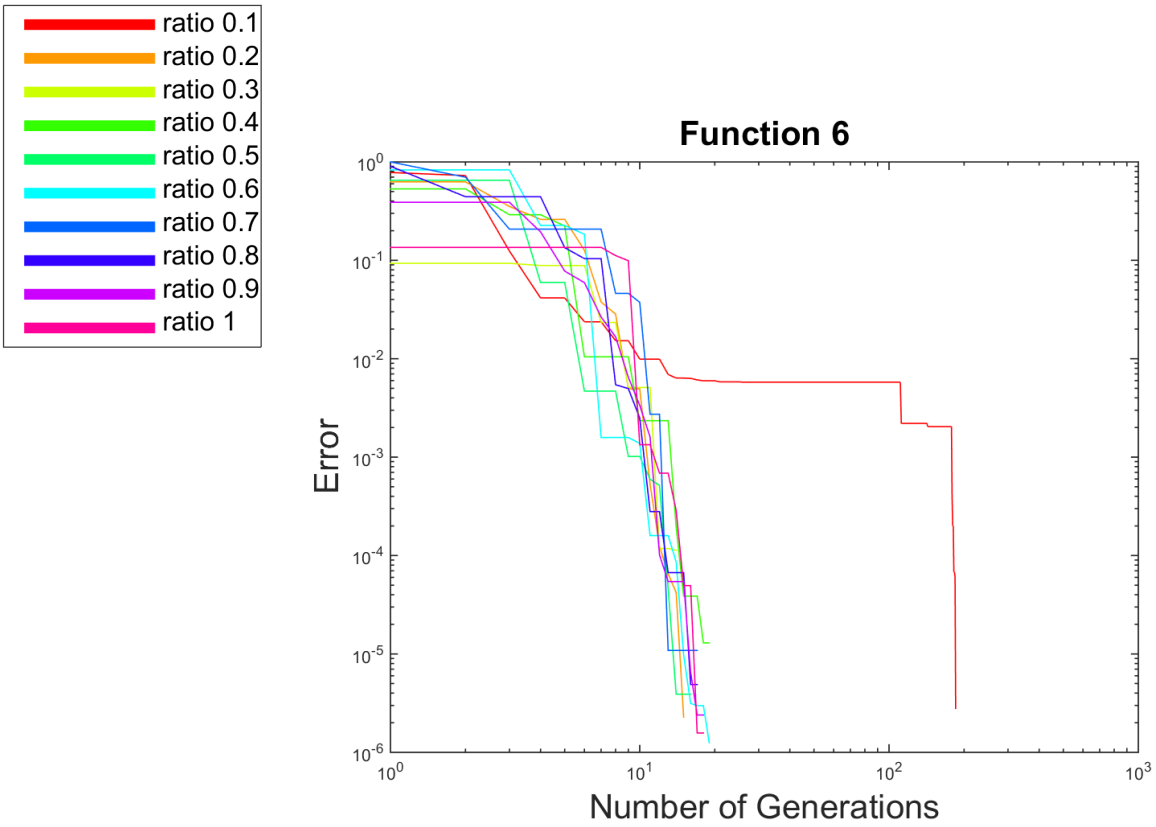


Figure 26.f

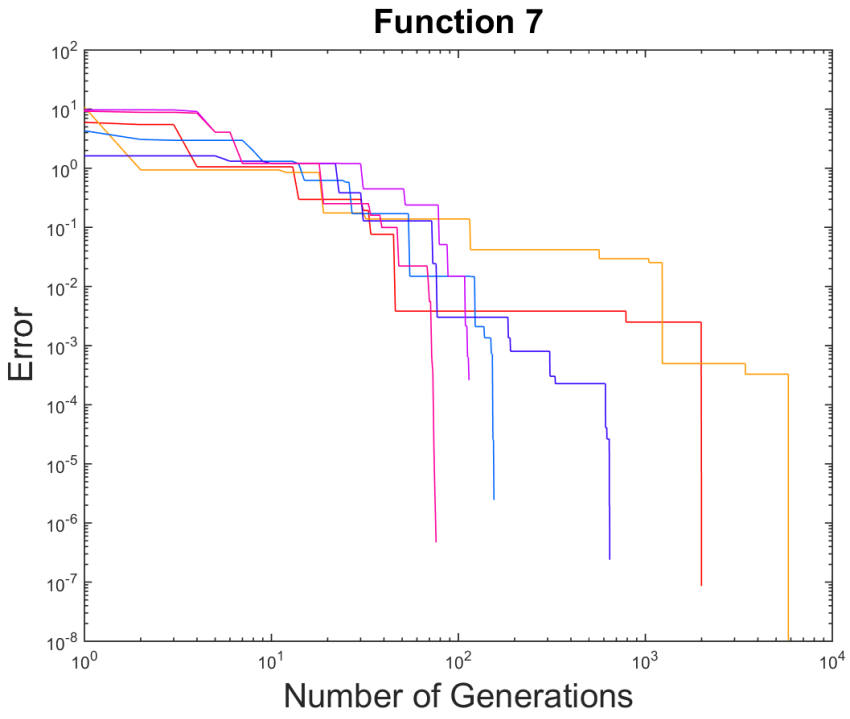


Figure 26.g

Figure 27: Spread in error for adjusting the ratio the size of xBest to the population size with crossover

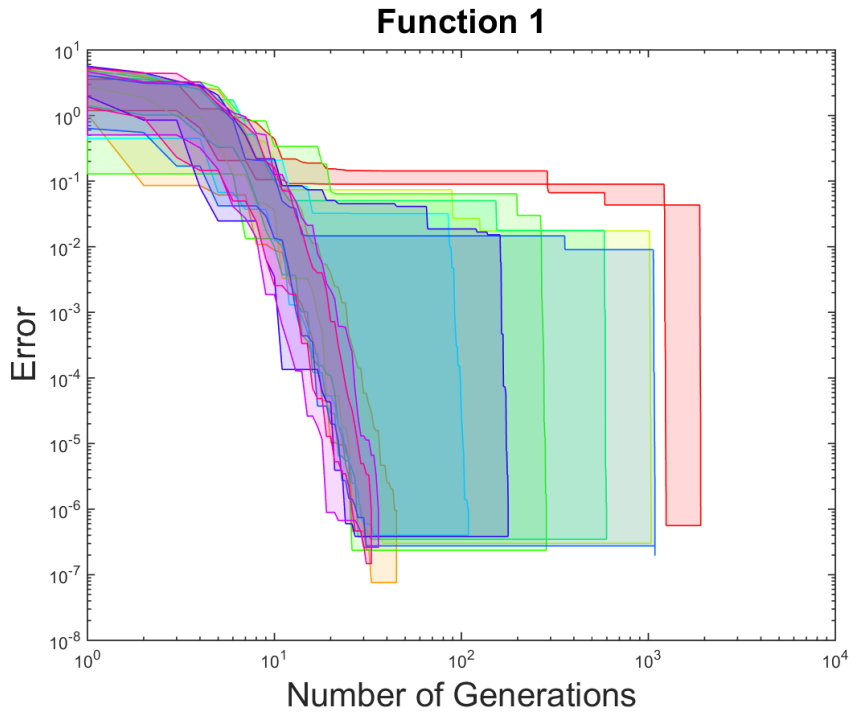


Figure 27.a

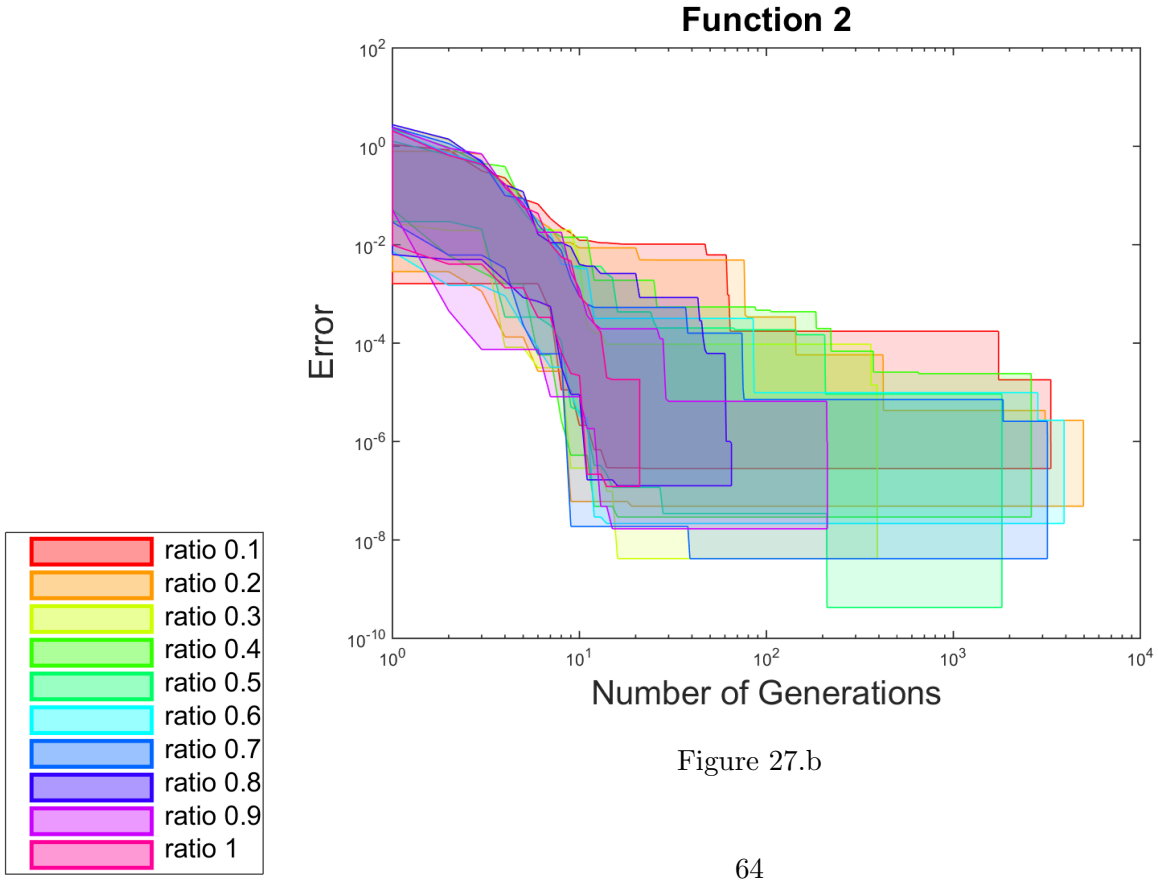


Figure 27.b

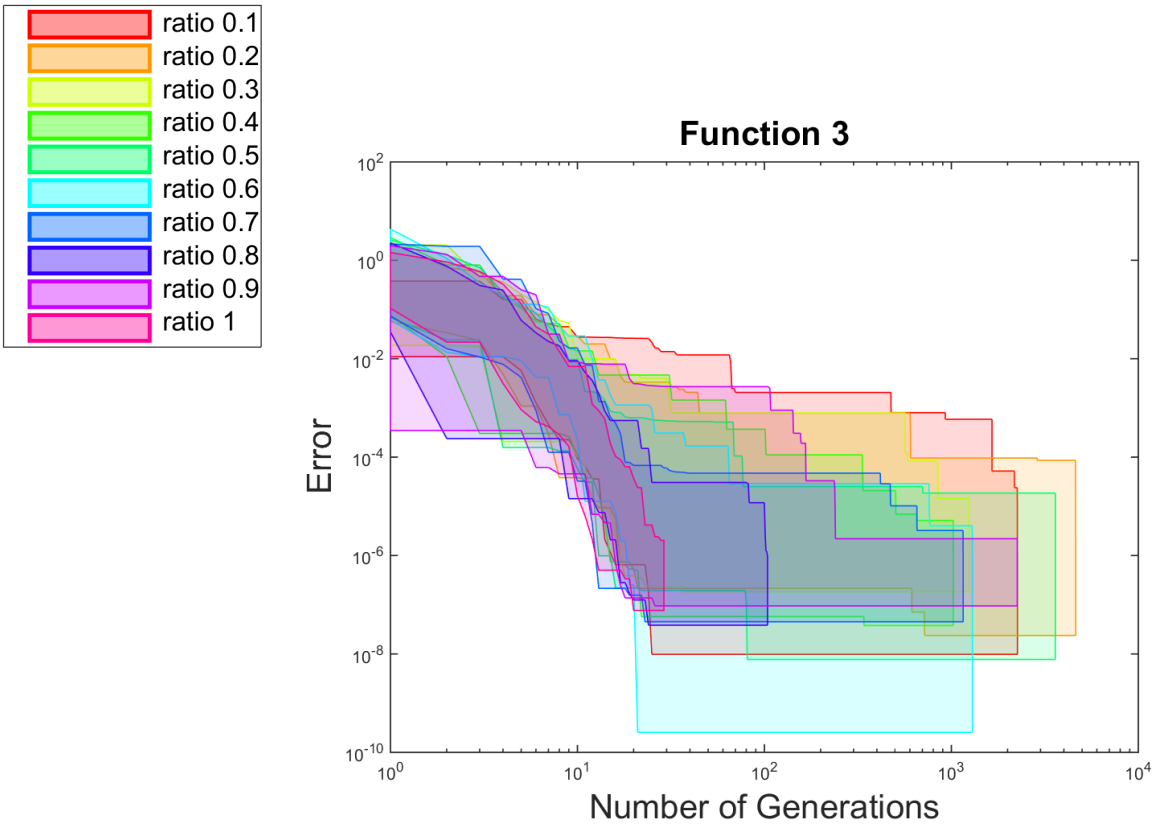


Figure 27.c

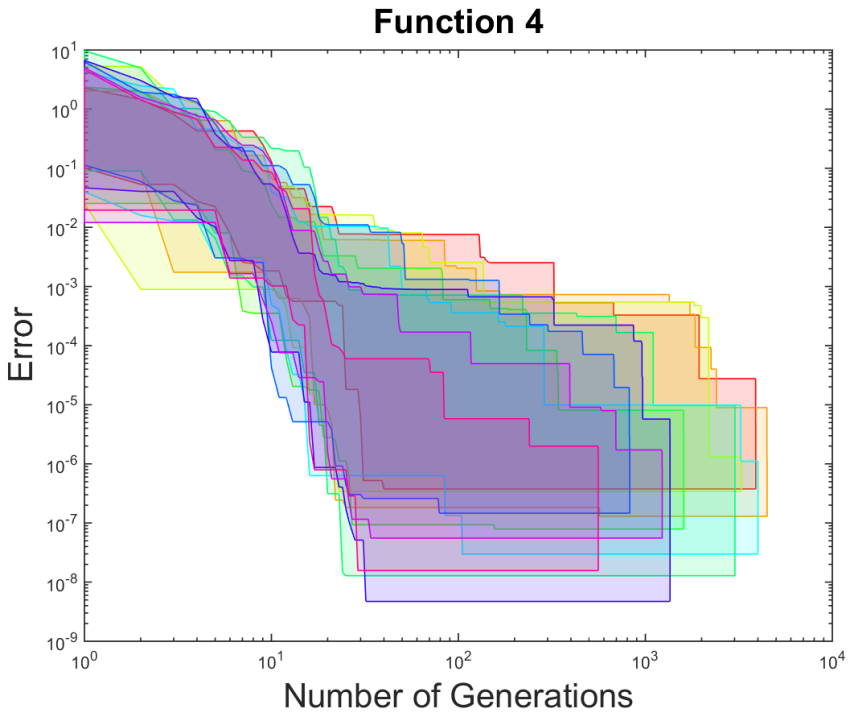


Figure 27.d

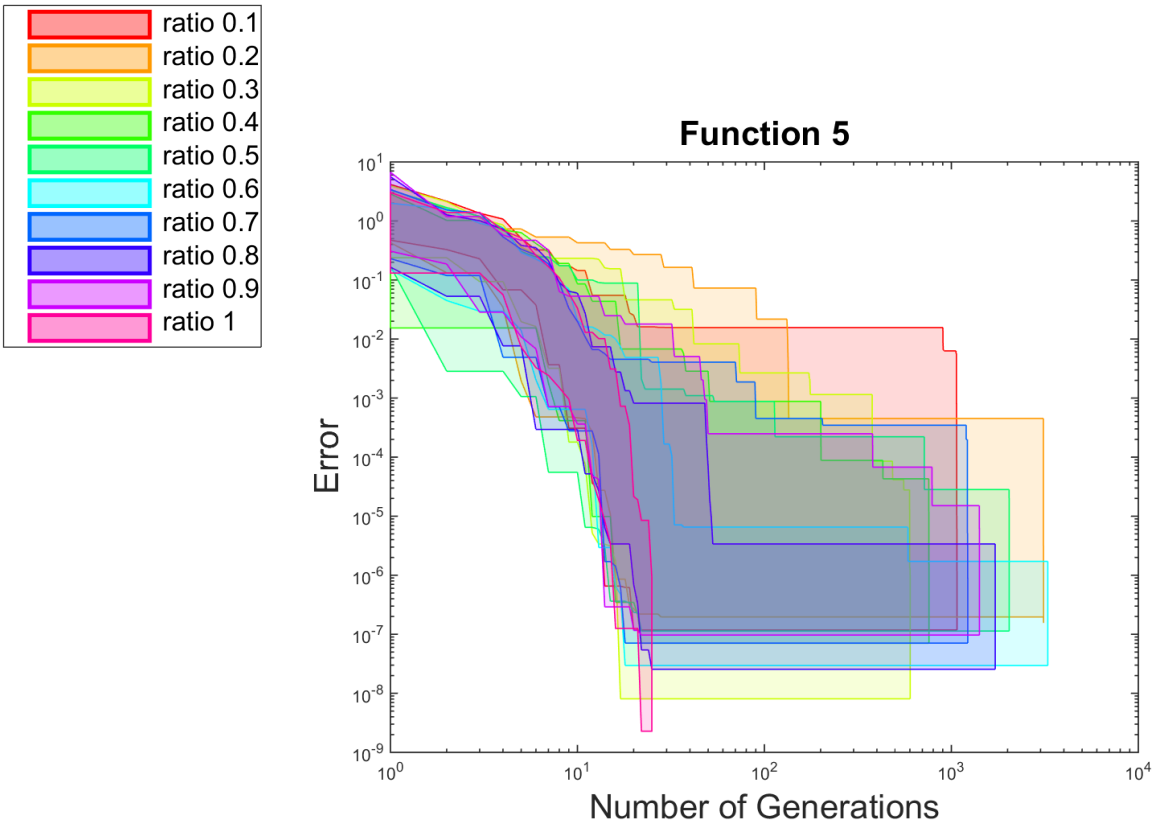


Figure 27.e

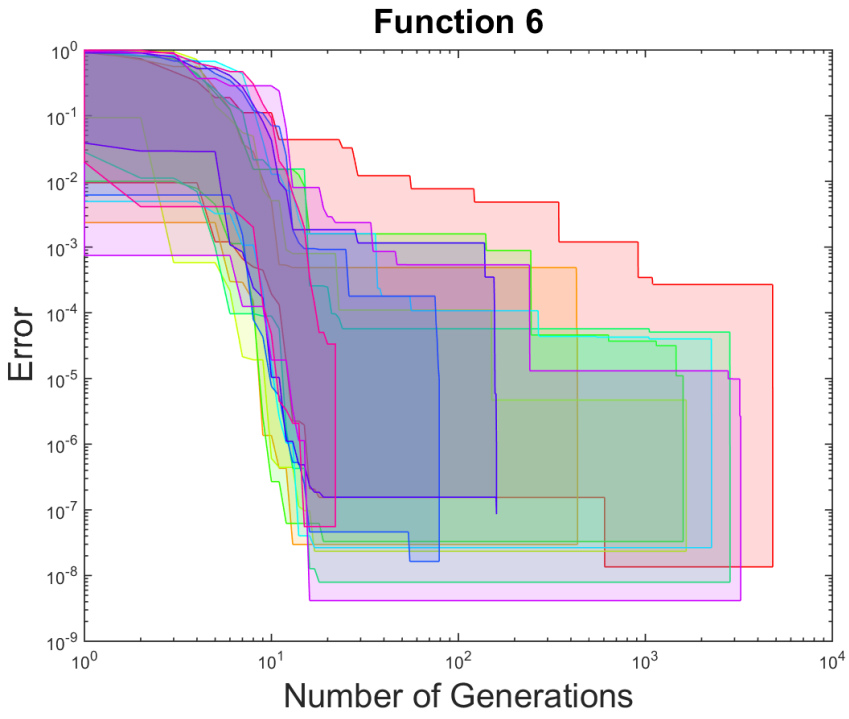


Figure 27.f

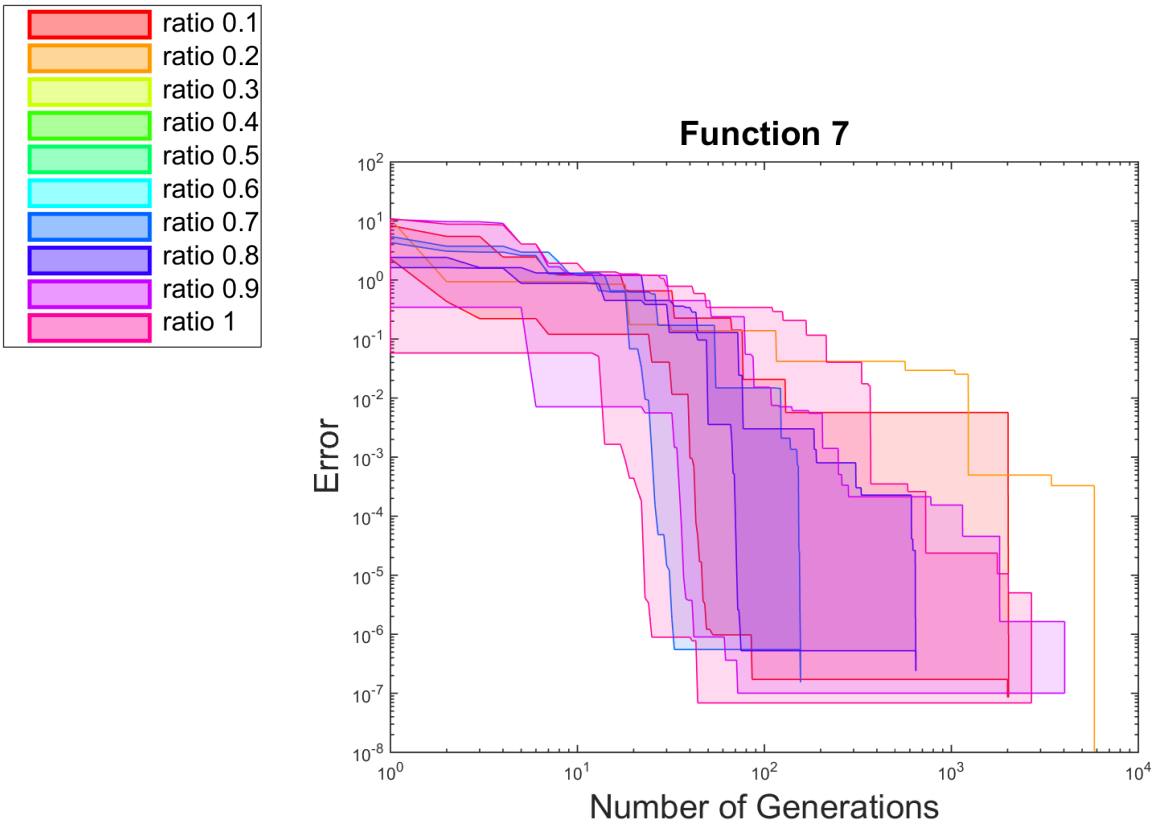


Figure 27.g

- From Figures 26, we see that the error decay is particularly slow for ratio 0.1 for most functions. However, for the other ratios for most functions, the error decay is quite similar.
- From Figure 27, we see that the spread of the errors in general is high and most ratios overlap each other, except for when ratio is 1 where the spread in most cases is very small compared to the other ratios.
- In conclusion we see that for crossover, the ratio of 1 gives the best results in terms of error decay.

3.8 Higher Dimensional Optimization

Here we study a function $f : [-5, 5]^{10} \to \mathbb{R}$, shown in Section 5.2.8, for our optimization algorithm. Here are some of the properties that we changed while testing our algorithm on this function.

- We used mutation and breeding together, we alternate between the two types of breeding in every generation.
- While studying using the algorithm on this function, convergence to the minimum was never achieved. The problem was with mutating or crossing over. Heuristically the chances of getting fitter elements after the jump or crossover is lowered as the

number of dimensions is higher. To get around this problem, I changed my algorithm so in each generation it will only mutate or crossover only in one dimension and cycle through the dimensions in different generations.

- We also had to adjust the temperature function to get the right decay for the jump:

$$\text{temperature}(n) = \frac{1}{(\log(\log(n)))^2}$$

After making these changes to the algorithm, we used the parameters in Table 9 to get results.

Table 9: Parameters used to get the data

Ratio of xBest to the population size	0.9
Ratio of number of elements to evaluate before starting a new generation to the population size	0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Number of workers	5
Repeats	10
Population size	50
Priority to the latest generation	0.5
Maximum Time	300 seconds
Tolerance	1×10^{-6}
Functions used	8
Breeding type	Mutation & Crossover

Figure 28: Results for adjusting the ratio of the number of elements needed to evaluate before starting a new generation to the population size for higher dimensional test function

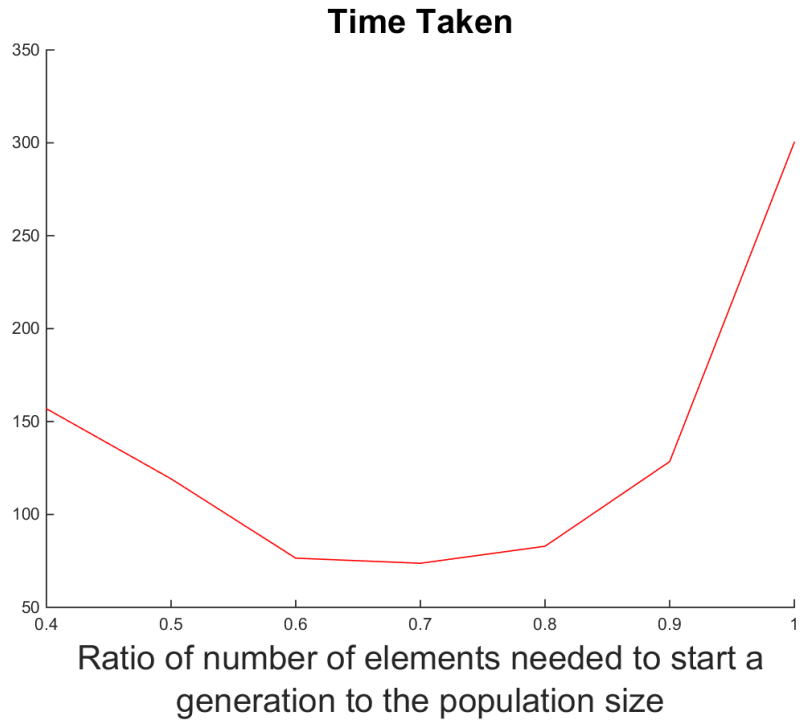


Figure 28.a

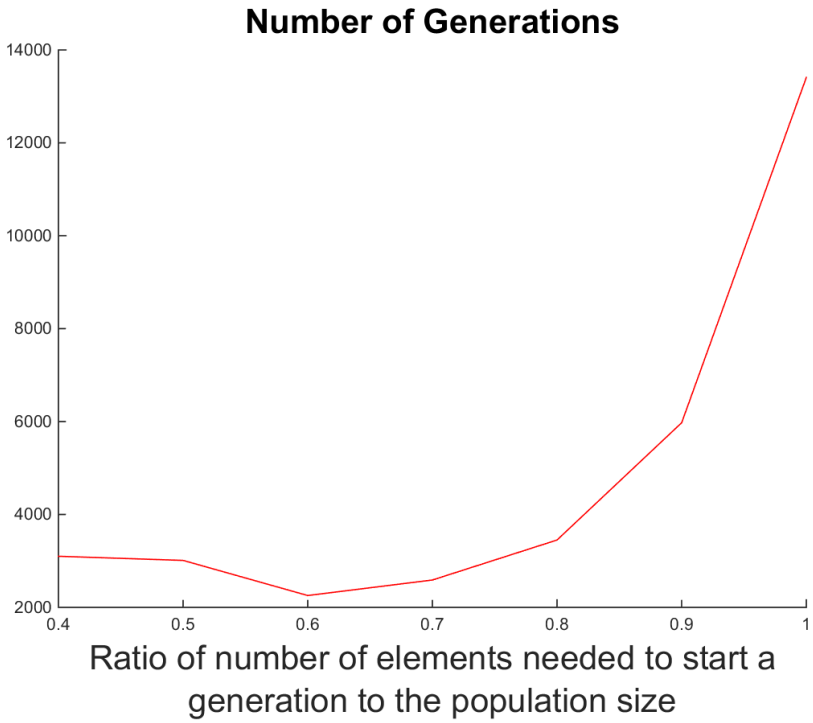


Figure 28.b

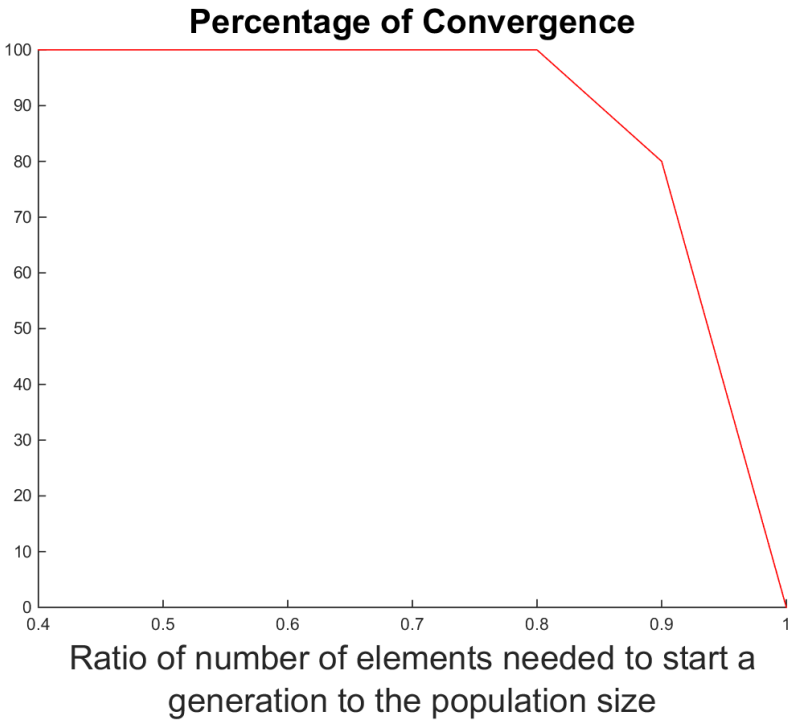


Figure 28.c

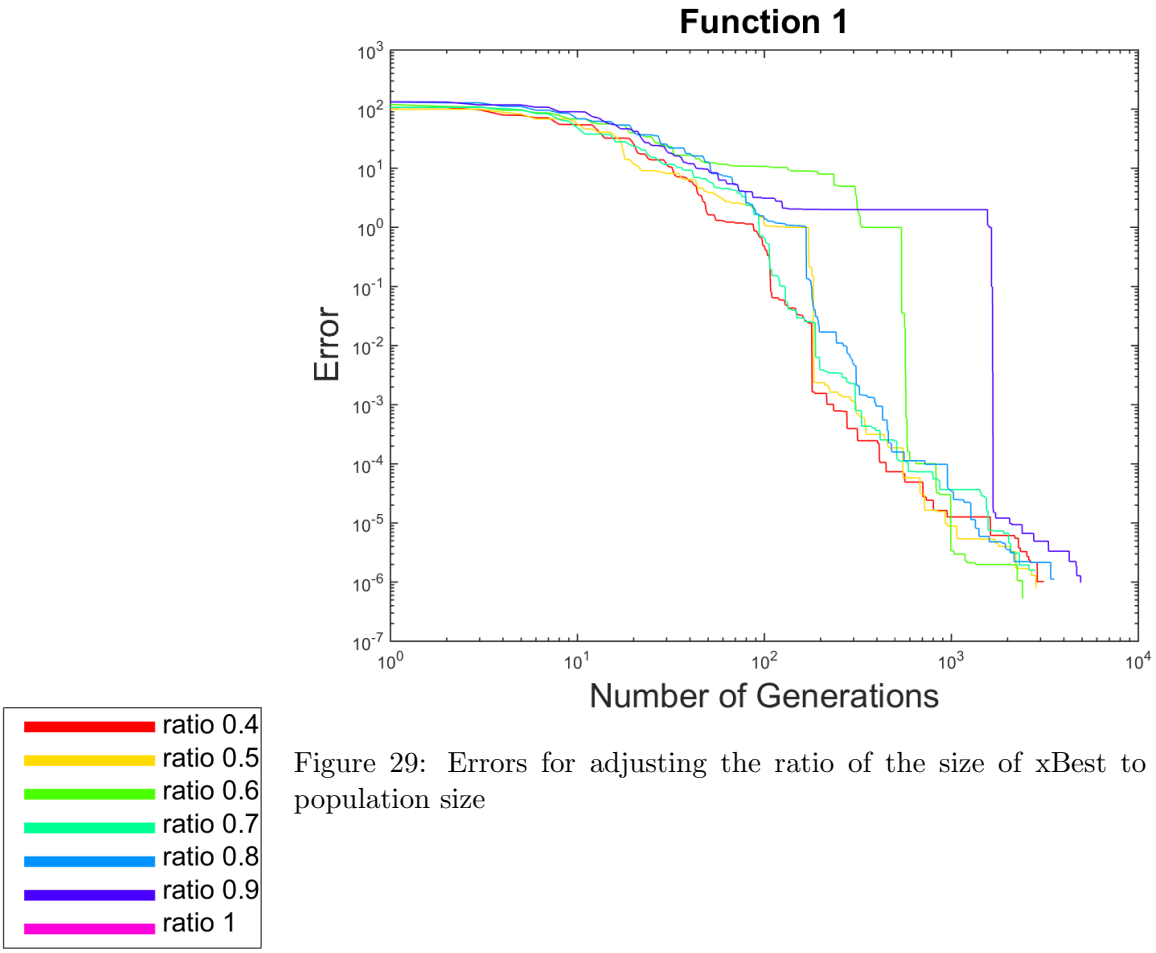


Figure 29: Errors for adjusting the ratio of the size of xBest to the population size

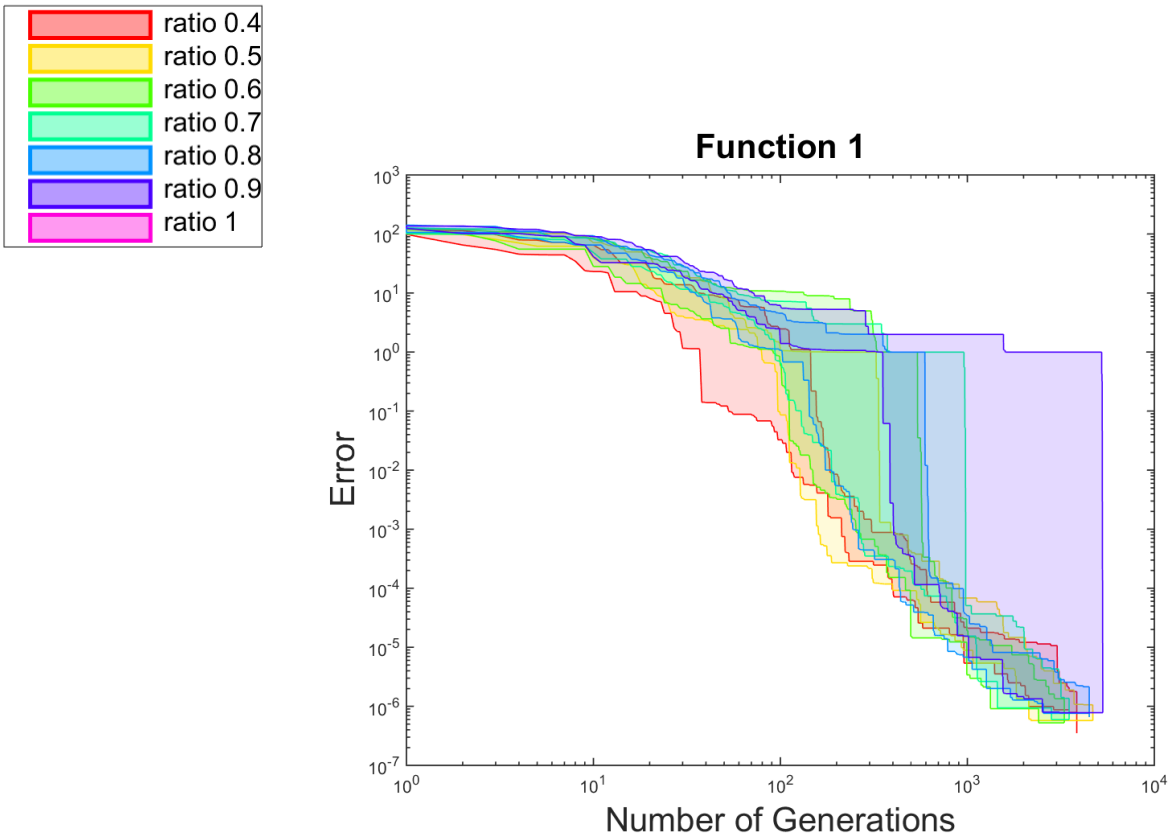


Figure 30: Spread in error for adjusting the ratio the size of xBest to the population size

- From Figure 28.c, we can immediately see that when the ratio is 1, we get no convergence. This gives evidence to use the asynchronous model that we have built for the algorithm.
- Figure 28.a displays that ratios between 0.6-0.8 give the fastest time for the algorithm to run.
- The number of repeats is low but we can still see a clear trend forming.
- Looking at Figure 30, we can see that the spread of the errors are fairly small except for when ratio is 0.9. This backs up the results that we obtained.
- Figure 29 shows that all ratios between 0.4-0.8 seem to have very similar error decays and the ratio of 0.9 has slower error decay.

4 Conclusion

4.1 Results

- We first saw that the speed up for the master-worker system was very good. This is because the time to evaluate the function is assumed to be much bigger than the other parts of the algorithm. Therefore, parallelising function evaluation means we are parallelising most of the algorithm in terms of time taken for the algorithm to run.
- Studying priority levels given to the latest generation, we found that priority levels between 0.4-0.6 would give us the best percentage of convergence to the minimum, fastest times to converge to the minimum and fastest error decays.
- For mutation, the ratios of the number of elements needed to be evaluated before starting the next generation to the population size between 0.4 and 0.6 gave us the best results in terms of time taken to converge to the minimum, the percentage of convergence and error decays. This provides support to the asynchronous model we are working with. The ratio of the size of the xBest to the size of the population of 0.5 gave the best results where the error decays were quite similar to error decays for ratios between 0.4-1.
- For crossover, both ratios we studied show that the best results are obtained when the ratios are equal to 1. This means that the asynchronous model is not good for this type of breeding.
- Finally, testing our algorithm on a higher dimensional function gave us a lot of information. First was that breeding on individual dimensions is a lot better than breeding with all dimensions at the same time. We saw that using both mutation and crossover worked well together. Finally, we saw that the temperature function is very important for convergence and the rate of convergence to the minimum. From the results we obtained, we found that the ratios of the number of elements needed to be evaluated before starting the next generation to the population size between 0.6 and 0.8 gave the best results. Again, this backs our asynchronous model.

4.2 Further Investigation That Can Be Made

- We would like to investigate into a variable temperature function that depends on the absolute difference between successive elements of xBest. This would be useful because we saw that some error decays were slow, for example Figure 17.e, and this would have been because the temperature function was not giving the jumps of the right size to get closer to the minimum.

- We could look into test functions which are continuous but not differentiable as our algorithm is not gradient based.
- Further investigate into breeding one dimension at a time rather than all dimensions at once as we saw in Section 3.8 that this worked better.
- Interesting results might be found if the ratio of the size of the xBest to the size of the population is greater than one but we would only make part of the population from the xBest and still have random points for the rest. We saw that we get bad convergence results to the minimum when the ratio is equal to 1 for mutation but we get the best convergence results for crossover. Having a ratio greater than one would mean for crossover we will have more points in our data set to fit a distribution, making the distribution potentially from accurate to sample from.

5 Appendix

5.1 Code

5.1.1 Main Script

MATLAB is use in implementing the algorithm from Section 2. Here is the main script for implementing the algorithm:

```
% Main Script

%% Initialization

% Start Timing for the whole code
tTotal = tic;

dim = 2;                              % Dimension of domain
bounds = [-5 5; -5 5];                % Bounds
popSize = 50;                         % Population size
totalGenNumber = 1e5;                 % Total number of generations
maxTime = 60;                         % Maximum time
f = @(x) x(1)^2+x(2)^2;               % Set the function here

% Ratio of number of elements to evaluate before starting the next
% generation to the population size
firstRatio = 0.5;

% Ratio of the size of xBest to the population size
fittestRatio = 0.6;

% Tolerance
tolerance = 1e-6;

% Priority given to the latest generation
priorityProbArray = 0.7;

% Numbers to represent information of elements
empty = 0;
filled = 1;
priority = 2;
busy = 3;
evaled = 4;

% Variable to represent convergence (1) or not (-1)
convg = 0;

% Actual number of elements from the respective ratios
```

```
39 fittestElements = round(fittestRatio*popSize);
40 firstElements = round(firstRatio*popSize);
41
42 % Cell arrays to hold data
43 x = cell(totalGenNumber+1,1);
44 fx = x;
45
46 spmd % Start parallel mode with communication
47
48     if labindex == 1 % Master's code
49
50         % Initalise arrays
51         for genNumber =1:totalGenNumber +1
52             x{genNumber +1} = zeros(dim,popSize);
53             fx{genNumber +1} = zeros(1,popSize);
54         end
55
56         % Initialise tracker array that keep track on information of
57         % elements
58         tracker = empty*ones(totalGenNumber +1,popSize);
59
60         % Initialise tracker array that keeps information about which
61         % worker has evaluated wwhich element
62         labTracker = tracker;
63
64         % Initalise variables for master
65         genNumber = 0;
66         priorityGenNumber = 0;
67         busyTracker = 0;
68         evaledTracker = 0;
69         evaledGenTracker = 0;
70         bestFound = 0;
71         endGenNumber = totalGenNumber;
72
73         % Initial population
74         x{genNumber+1} = popGen(dim,bounds,popSize,genNumber,...
75             totalGenNumber,fittestElements,0);
76         tracker(genNumber+1,:) = filled*ones(1,popSize);
77
78         % fxBest given big numbers to start with
79         xBest = x{genNumber+1}(:,1:fittestElements);
80         fxBest = 1e5*ones(1,fittestElements);
81
82         % Variable to show all workers are finished or not
83         done = zeros(1,numlabs-1);
84
85         % Start while loop to send/recv information
86         while isequal(done,ones(1,numlabs-1)) ~= 1
87
```

```
88              % Receive message from any worker
89              temp = labReceive('any',0);
90
91              % If 0 then worker requesting an element for evaluation
92              if temp{1} == 0
93
94                  labIdx = temp{2};
95
96                  % If no elements left then tell worker he is done
97                  if (busyTracker + evaledTracker == numel(tracker)...
98                          || bestFound == 1 || toc(tStart) > maxTime)
99
100                     labSend({'done'},labIdx,1);
101
102                     % If maximum time is reached or maximum number of
103                     % generations reached then we have not converged
104                     if (toc(tStart) > maxTime ||...
105                             busyTracker + evaledTracker == numel(tracker))
106                         convg = -1;
107                         endGenNumber = priorityGenNumber;
108                     end
109
110
111                 % Otherwise send an element some work
112                 else
113
114                     % Find elements for evaluation
115                     [genNumber,popNumber] = indexOfElement(tracker,...
116                         priorityProb,filled,priorityGenNumber);
117
118                     % If we cannot find an element then ask worker to wait
119                     if strcmp(genNumber,'wait') == 1
120                         labSend({'wait'},labIdx,1);
121
122                     % If we found then element then send it
123                     else
124                         labSend({genNumber,popNumber,...
125                             x{genNumber+1}(:,popNumber)},labIdx,1);
126
127                         % Update tracker
128                         tracker(genNumber+1,popNumber) = busy;
129                         labTracker(genNumber+1,popNumber) = labIdx;
130                         busyTracker = busyTracker +1;
131                     end
132                 end
133
134             % If 1 then worker sending evaluated element back
135             elseif temp{1} == 1
136
```

```
137                 % Check what element is received and store the results
138                 genNumber = temp{3};
139                 popNumber = temp{4};
140                 fx{genNumber+1}(popNumber) = temp{5};
141 
142                 % Update xBest
143                 if priorityGenNumber ~= 0
144                     [xBest,fxBest] = updateBest(xBest,fxBest,...
145                         x{genNumber+1}(:,popNumber),temp{5},...
146                         fittestElements);
147                 end
148 
149                 % Check if tolerance level is reached
150                 if abs(fx{genNumber+1}(popNumber)-fmin) < tolerance
151                     bestFound = 1;
152                     endGenNumber = priorityGenNumber;
153                     % We have converged
154                     convg = 1;
155                 end
156 
157                 % Update tracker
158                 tracker(genNumber+1,popNumber) = evaled;
159                 evaledTracker = evaledTracker +1;
160                 busyTracker = busyTracker -1;
161                 evaledGenTracker = evaledGenTracker +1;
162 
163                 % Check how many elements have beeen evaluated in this
164                 % generation
165                 idx = find(tracker(genNumber+1,:) == evaled);
166                 counter = numel(idx);
167 
168                 % If we have evaluated enough elements then start next
169                 % generation
170                 if evaledGenTracker == firstElements &&...
171                         genNumber < totalGenNumber
172 
173                     x{genNumber+1 +1} = popGen(dim,bounds,popSize,...
174                         genNumber+1,totalGenNumber,fittestElements,...
175                         xBest);
176 
177                     % Update tracker
178                     tracker(genNumber+1 +1,:) = filled*ones(1,popSize);
179                     priorityGenNumber = genNumber +1;
180                     evaledGenTracker = 0;
181                 end
182 
183             % If 2 then the worker has been told that he is done and the
184             % master knows that too
185             elseif temp{1} == 2
```

```
186
187                 labIdx = temp{2};
188                 % Update done
189                 done(labIdx-1) = 1;
190
191             end % Type of message received
192         end % Master while loop
193
194     else    % Worker's code
195
196         % Variable to show that worker is done or not
197         done = 0;
198
199         % While loop to send and receive information
200         while done ~= 1
201
202             % Send message to master to asking for an element
203             labSend({0,labindex},1,0);
204
205             % Receive message from master with an element
206             temp = labReceive(1,1);
207
208             % If message is 'done' then the worker is finished
209             if strcmp(temp{1},'done') == 1
210
211                 % Send final message saying the worker is done
212                 labSend({2,labindex},1,0);
213                 done = 1;
214
215             % Otherwise work is received which needs to be done
216             elseif strcmp(temp{1},'wait') == 1
217                 continue;
218
219             else
220
221                 % Information from the message
222                 genNumber = temp{1};
223                 popNumber = temp{2};
224                 x = temp{3};
225
226                 % Evaluate the element where 'f' is the function we are
227                 % evaluating
228                 fx = funcEval(dim,1,f,x);
229
230                 % Send the evaluted element back to the master
231                 labSend({1,labindex,genNumber,popNumber,fx},1,0);
232
233             end
234         end % Worker while loop
```

```
235     end % If master/worker
236 end % SPMD
237
238 % Total time taken
239 tTotal = toc(tTotal);
240 disp(tTotal);
```

5.1.2 Population Generation

Here is the code for generating a new population:

```
1  function [newGen] = popGen(dim,bounds,popSize,genNumber,...
2      fittestElements,xBest)
3
4      if genNumber == 0
5          % If inital population, take uniform random points within bounds
6          % Potentially can be parallelised
7          newGen = randPopGen(dim,bounds,popSize);
8      else
9
10         % Make the first part of the population from xBest
11         newGen(:,1:fittestElements) = breed(dim,bounds,fittestElements,...
12             genNumber,xBest);
13
14         % For the rest of the population get uniform random points
15         % within bounds
16         if fittestElements < popSize
17             newGen(:,fittestElements+1:popSize) = randPopGen(dim,bounds,...
18                 popSize-fittestElements);
19         end
20
21         % Randomise population
22         idx = randperm(popSize);
23         newGen = newGen(:,idx);
24
25     end
26 end
```

5.1.3 Random Population Generation

Here is the code for generating a uniformly random population:

```
1 function [x] = randPopGen(dim,bounds,popSize)
2
```

```
    % Calculate the size of each interval
    space = abs(bounds(:,2) - bounds(:,1));

    % Create uniform random number between 0 and 1 then multiply by the
    % size of the interval then add the left side of the interval
    x = kron(bounds(:,1),ones(1,popSize)) +...
    kron(space,ones(1,popSize)).*rand(dim,popSize);

end
```

5.1.4 Breeding

Here is the code for breeding in generating the population from xBest:

```
function [x] = breed(breedtype,dim,bounds,popSize,genNumber,xBest)

    % Initialise output
    x = zeros(dim,popSize);

     % Sample from standard normal distribution
     r = randn(dim,popSize);

    % If breeding type is crossover
    if breedtype == 1

        % Mean and standard deviation for estimating Gaussian distribution
        mu = mean(xBest,2);
        C = std(xBest,0,2);

        % Translate and rescale to make the same for the estimated Gaussian
        % distribution and check if in bounds, if not then replace with
        % bound
        for i = 1:dim
            for j = 1:popSize
                x(i,j) = mu(i) + C(i)*r(i,j);
                if x(i,j) > bounds(i,2)
                    x(i,j) = bounds(i,2);
                elseif x(i,j) < bounds(i,1)
                    x(i,j) = bounds(i,1);
                end
            end
        end

    % If breeding type is mutation
    elseif breedtype == 0

        % Function for temperature
```

```
34          g = @(genNumber) 1/(10*genNumber);
35
36          % Rescaling constant from temperature function
37          c = g(genNumber);
38
39          % Loop over all the individual components and add the jumps and
40          % then check for bounds, if out of bounds then replace with bound
41          for i = 1:dim
42              for j = 1:popSize
43                  x(i,j) = xBest(i,j) + c*r(i,j);
44                  if x(i,j) > bounds(i,2)
45                      x(i,j) = bounds(i,2);
46                  elseif x(i,j) < bounds(i,1)
47                      x(i,j) = bounds(i,1);
48                  end
49              end
50          end
51      end
52  end
```

5.1.5 Picking Element to Evaluate

Here is the code for picking a generation and then an element in that generation for evaluation:

```
1   function [genNumber,popNumber] = indexOfElement(tracker,priorityProb,...
2       filled,priorityGenNumber)
3
4       % Initialise N for while loop
5       N = priorityGenNumber+1;
6
7       % Sample from geometric distribution with given priorityProb until it
8       % is between 0 and prioritygenNumber
9       while N >priorityGenNumber
10          N = geornd(priorityProb);
11      end
12
13      % Find an element in the generation priorityGenNumber+1-N
14      idx = find(tracker(priorityGenNumber+1-N,:) == filled,1);
15
16      % If there is none, send 'wait'
17      if isempty(idx) == 1
18           genNumber = 'wait';
19           popNumber = 0;
20      % Otherwise send the index of the element
21      else
22           genNumber = priorityGenNumber - N;
```

```
23            popNumber = idx;
24        end
25
26    end
```

5.1.6 Function Evaluation

Here is the code for the function evaluation, the parameters include the element(s) you are trying to evaluate and the minimising function. Instead you can adjust this so that it have the minimising function defined within the code.

```
1   function [fx] = funcEval(dim,popSize,f,x)
2
3       % Initialize array
4       fx = zeros(1,popSize);
5
6       % For loop to evaluate the function over all of population
7       for i = 1:popSize
8           fx(i) = f(x(:,i));
9       end
10
11  end
```

5.2 Test Functions

The following are the test functions with the minimisers shown as a red star marker.

5.2.1 Ackley's Function: $\mathbb{R}^2 \to \mathbb{R}$

$$f(x,y) = -20\exp\left(-0.2\sqrt{0.5\left(x^2+y^2\right)}\right) - \exp\left(0.5\left(\cos\left(2\pi x\right)+\cos\left(2\pi y\right)\right)\right) + e + 20$$

Thorny function with minimum at $(0,0)$ and $D = [-5,5] \times [-5,5]$.

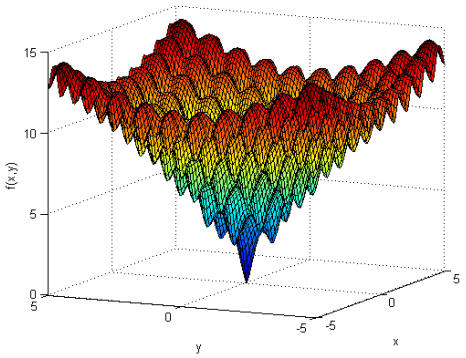


Figure 31: Ackley's Function

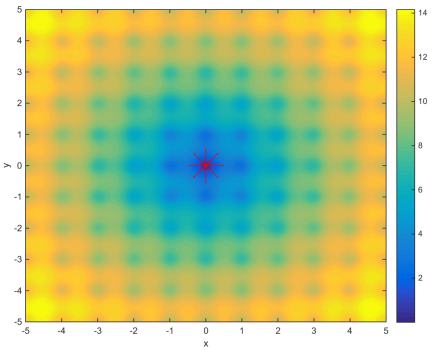


Figure 32: Ackley's Function Contour

5.2.2 Sphere Function $\mathbb{R}^2 \to \mathbb{R}$

$$f(x,y) = x^2 + y^2$$

Very smooth function where $D = \mathbb{R}^2$ and minimum at $(0,0)$

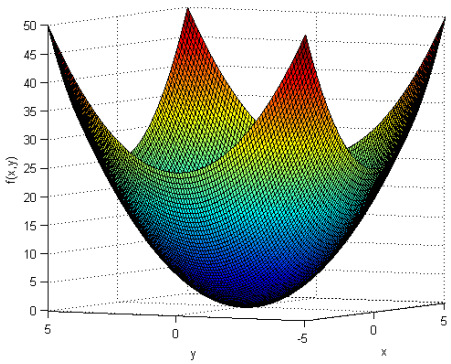


Figure 33: Sphere Function

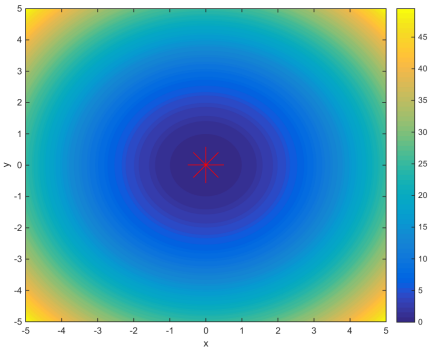


Figure 34: Sphere Function Contour

5.2.3 Rosenbrock Function $\mathbb{R}^2 \to \mathbb{R}$

$$f(x, y) = (1 - x)^2 + \left(y - x^2\right)^2$$

Minimum at $(1, 1)$, $D = [-5, 5] \times [-5, 5]$.

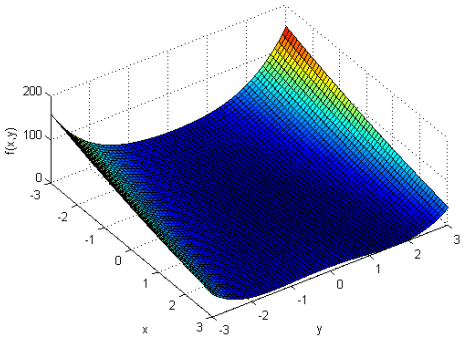


Figure 35: Rosenbrock Function

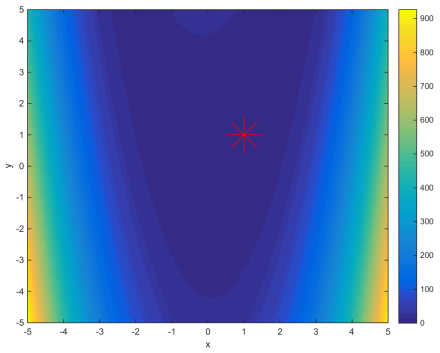


Figure 36: RosenBrock Function Contour

5.2.4 Beale's Function $\mathbb{R}^2 \to \mathbb{R}$

$$f(x, y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2$$

Minimum at $(3, 0.5)$, $D = [-5, 5] \times [-5, 5]$.

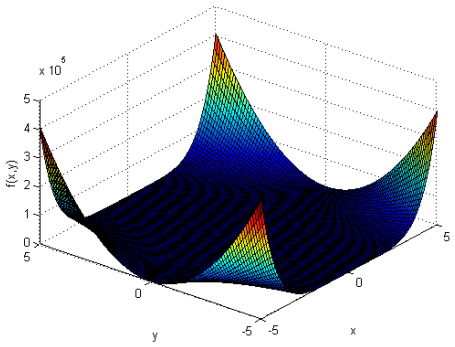


Figure 37: Beale's Function

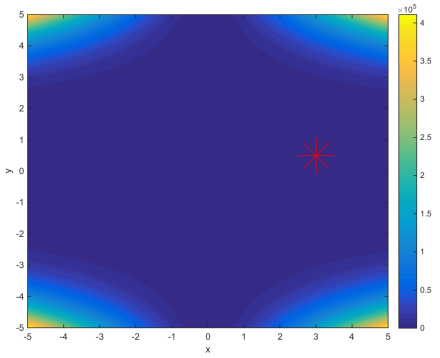


Figure 38: Beale's Function Contour

5.2.5 Levi Function $\mathbb{R}^2 \to \mathbb{R}$

$$f(x,y) = \sin^2(3\pi x) + (x-1)^2\left(1+\sin^2(3\pi y)\right) + (y-1)^2\left(1+\sin^2(2\pi y)\right)$$

Thorny function with minimum at $(1,1)$, $D = [-5,5] \times [-5,5]$.

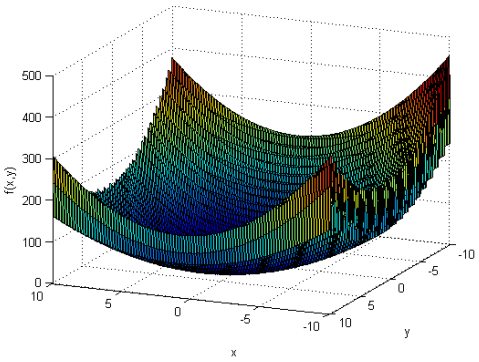


Figure 39: Levi Function

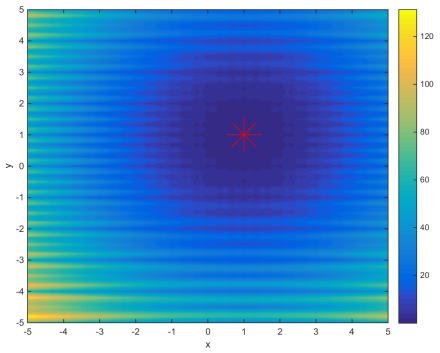


Figure 40: Levi Function Contour

5.2.6 Easom Function $\mathbb{R}^2 \to \mathbb{R}$

$$f(x,y) = -\cos(x)\cos(y)\exp\left(-\left((x-\pi)^2 + (y-\pi)^2\right)\right)$$

Minimum at (π,π), $D = [-5,5] \times [-5,5]$.

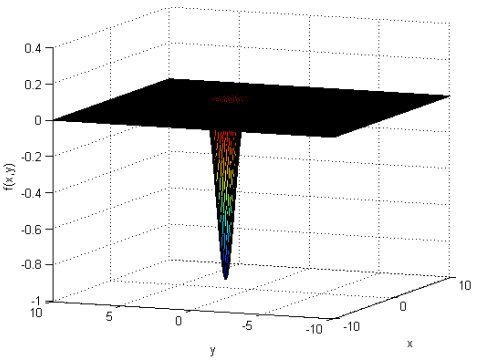


Figure 41: Easom Function

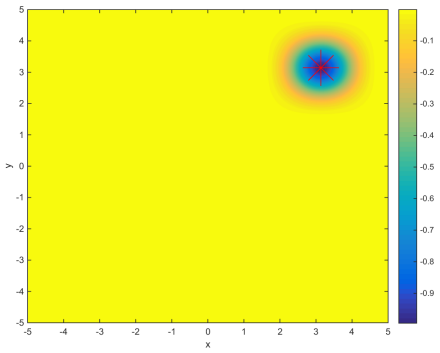


Figure 42: Easom Function Contour

5.2.7 Holder Table Function $\mathbb{R}^2 \to \mathbb{R}$

$$f(x,y) = -\left|\sin(2x)\cos(2y)\exp\left(\left|1 - \frac{\sqrt{(2x)^2+(2y)^2}}{\pi}\right|\right)\right|$$

Minimums at $(\alpha,\beta),(-\alpha,\beta),(\alpha,-\beta),(-\alpha,-\beta)$ where $\alpha = 4.02751$ and $\beta = 4.832295$. $D = [-5,5] \times [-5,5]$.

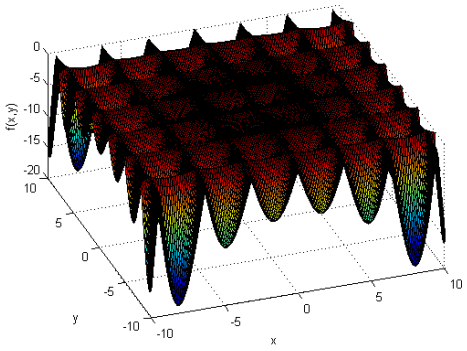


Figure 43: Holder Table Function

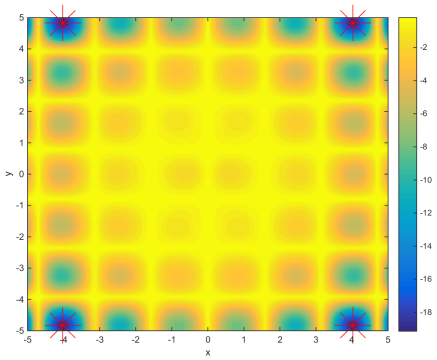


Figure 44: Holder Table Function Contour

5.2.8 Rastrigin Function $\mathbb{R}^d \to \mathbb{R}$

$$f(x_1,\ldots,x_d) = 10d + \sum_{i=1}^{d}\left(x_i^2 - 10\cos\left(2\pi x_i\right)\right)$$

Minimum at $(0,\ldots,0)$. $D = [-5,5]^d$.

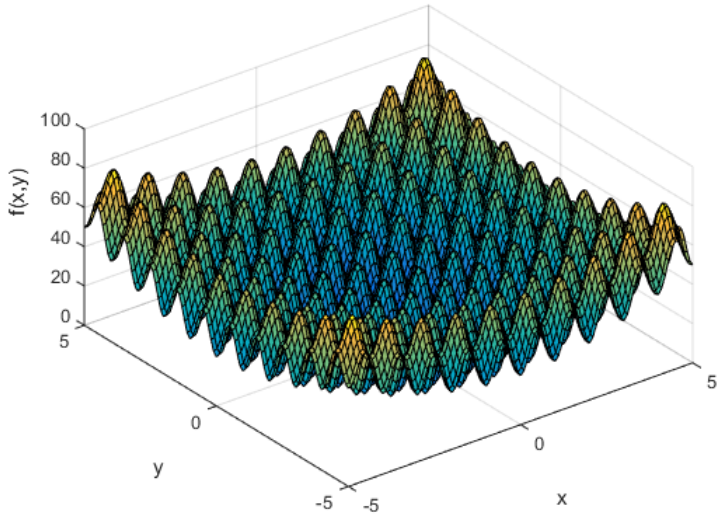


Figure 45: Rastrigin Function For d=2

5.2.9 Sample Paths for the Test Functions

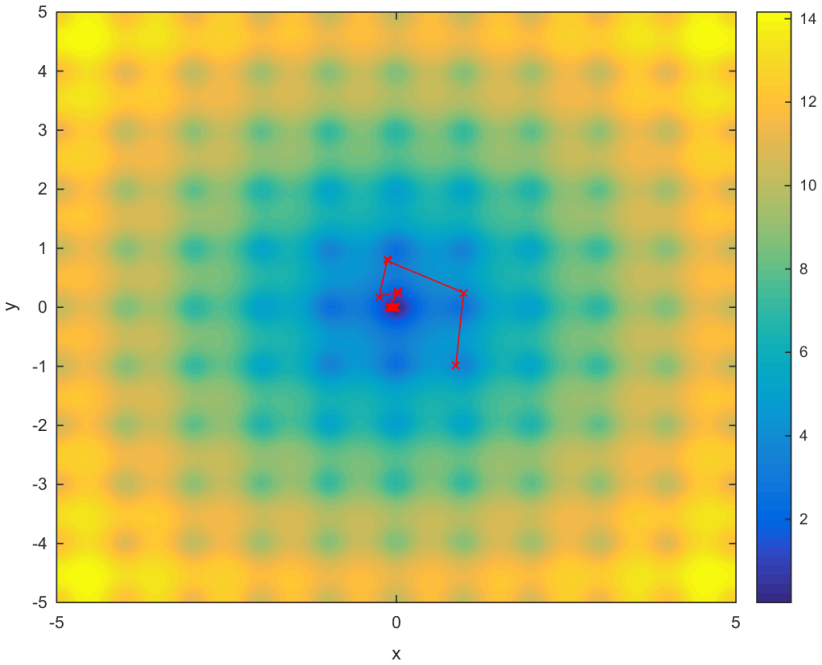


Figure 46: Function 1

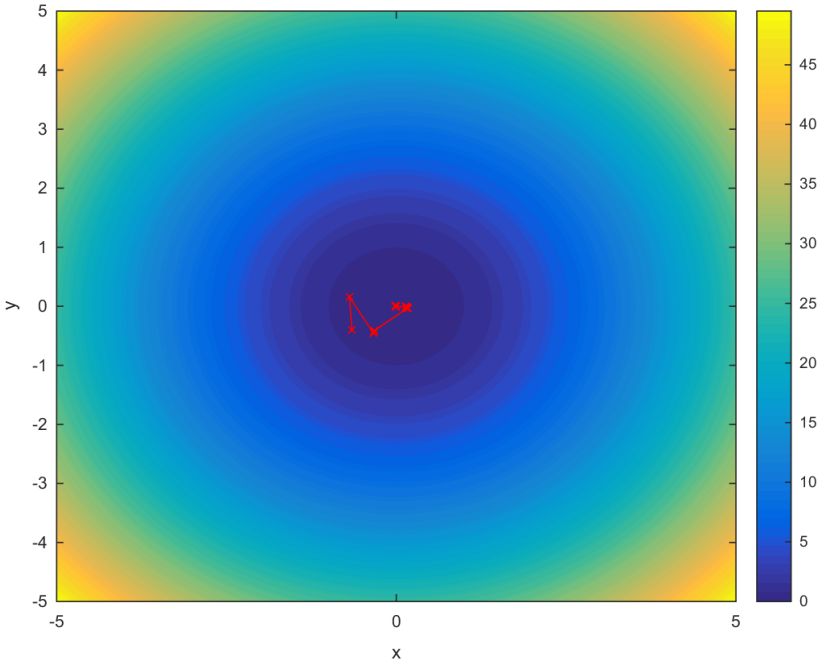


Figure 46.a: Function 2

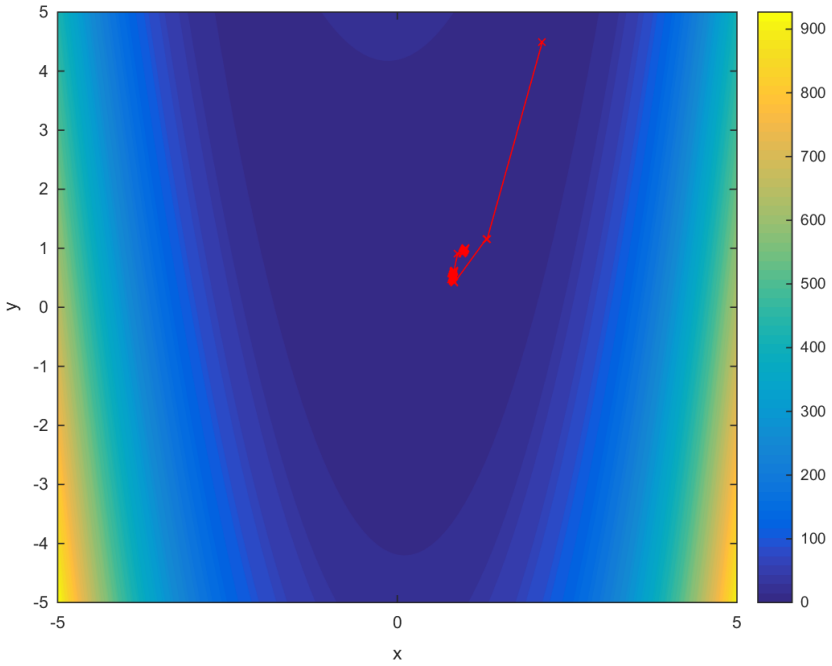


Figure 46.b: Function 3

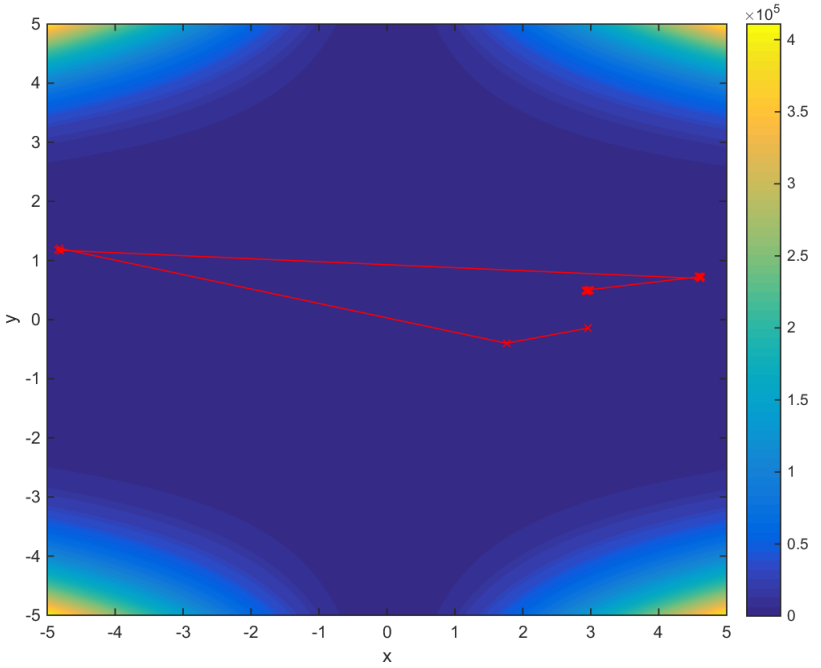


Figure 46.c: Function 4

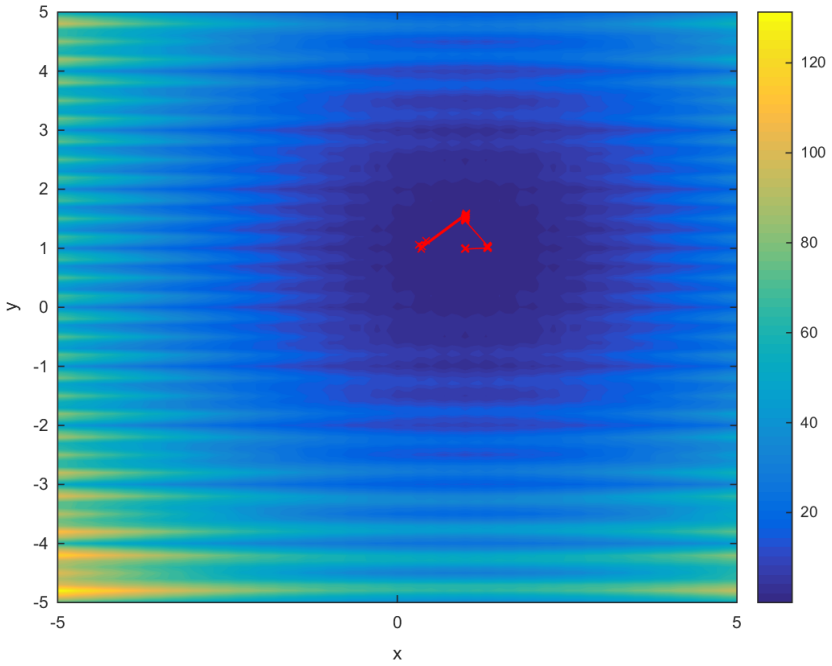


Figure 46.d: Function 5

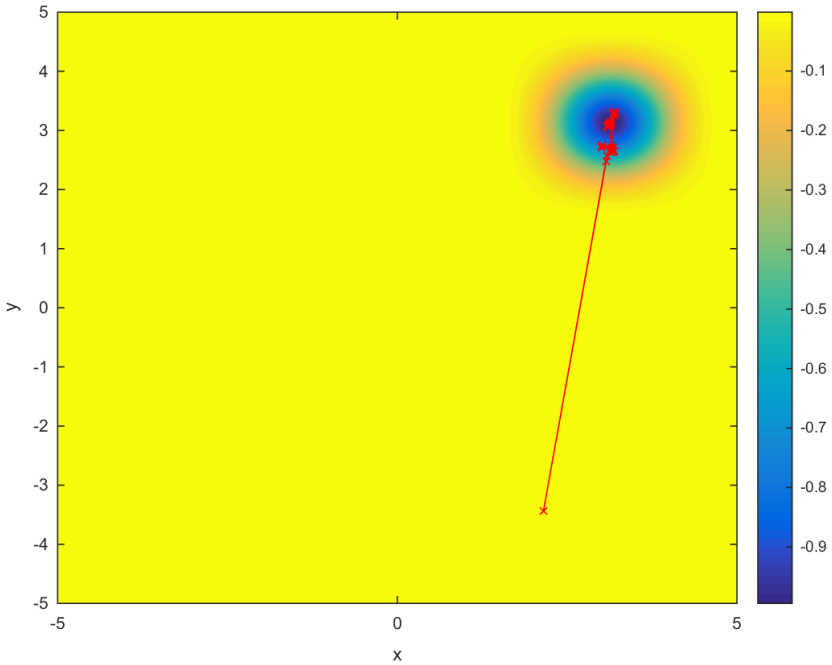


Figure 46.e: Function 6

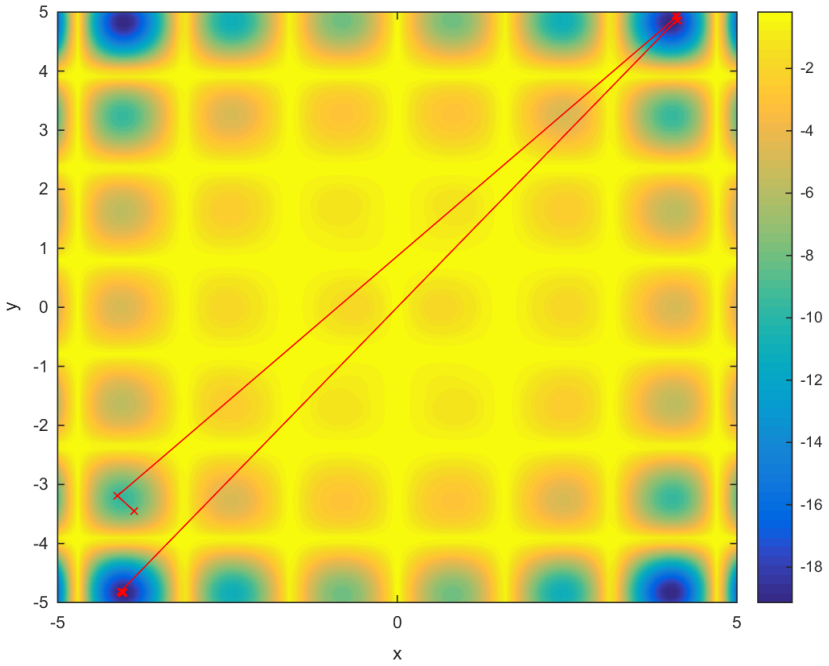


Figure 46.f: Function 7

Bibliography

[1] Z.Michalewicz, *Genetic Algorithms + Data Structures = Evolutionary Programs*, Springer, 1992, 17-18.

[2] G.Luque, E.Alba, *Parallel Genetic Algorithms: Theory and Real World applications*, Springer, 2011, 20-21.

[3] W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 2007, 842-847.