
# Competetive Coevolution in Problem Design and Metaheuristical Parameter Tuning

Guðmundur Einarsson

Faculty of Physical Sciences
University of Iceland
2014

# COMPETETIVE COEVOLUTION IN PROBLEM DESIGN AND METAHEURISTICAL PARAMETER TUNING

Guðmundur Einarsson

60 ECTS thesis submitted in partial fulfillment of a
*Magister Scientiarum* degree in Statistics

Advisors
Gunnar Stefánsson
Thomas Philip Runarsson

Faculty Representative
Gunnar Stefánsson

External Examiner
Sven Þ. Sigurðsson

Faculty of Physical Sciences
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, May 2014

Competetive Coevolution in Problem Design and Metaheuristical Parameter Tuning
Competitive Coevolution Predator Prey Scheme
60 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Statistics



Physical Sciences
School of Engineering and Natural Sciences
University of Iceland
Dunhaga 5
107 Reykjavík, Reykjavik
Iceland

Telephone: 525 4000

Bibliographic information:
Guðmundur Einarsson, 2014, Competetive Coevolution in Problem Design and Metaheuristical Parameter Tuning, M.Sc. thesis, Faculty of Physical Sciences, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík
Reykjavik, Iceland, May 2014

*Dedicated to my parents*

# Abstract

The Marine Research Institute of Iceland has over the last 20 years developed and used the program `gadget` for modeling of the marine ecosystem around Iceland. The estimation of parameters in this program requires constrained optimization in a continuous domain. In this thesis a coevolutionary algorithm approach is developed to tune the optimization parameters in `gadget`. The objective of the coevolutionary algorithm is to find optimization parameters that both make the optimization methods in `gadget` more robust against poorly chosen starting values and tries to reduce the computation time while maintaining convergence. This may also ease the tuning of optimization parameters for new users and may reveal other local optima in the likelihood, which may give hint of model misspecification. The algorithm is tested on functions that have similar characteristics as the log-likelihood functions in `gadget` and some results shown for the case of modeling haddock.

# Útdráttur

Hafrannsóknarstofnun hefur á síðustu 20 árum staðið að hönnun og notkun stofnmatsforritsins `gadget`. Þetta forrit útfærir vistfræðileg tölfræðilíkön og notar stofnunin það við gerð líkana fyrir sjávarlífið á grennd um Ísland. Í forritinu er stikar metnir með bestunaraðferðum í samfelldu afmörkuðu rúmi. Í þessari ritgerð er þróunaralgrím útfært sem þjálfar bestunaraðferðirnar í `gadget` til að takast á við erfið byrjunargildi. Erfið byrjunargildi geta stafað út af staðbundnu útgildi eða slæmu ástandi fallsins. Útkoma algrimsins getur einfaldað nýjum notendum að beita bestunaraðferðunum í `gadget` og birtir mögulega önnur staðbundin útgildi sem gefa vísbendingu um að líkanið sé ekki rétt valið. Reikniritið er prófað á föll sem hafa svipaða eiginleika og lograsennileikafallið í `gadget` og eru niðurstöður sýndar úr einföldu ýsulíkans dæmi.

# Preface

The work presented in this thesis is the fruit of my research during my latter year as an M.Sc. student at the University of Iceland in fulfilment of the requirements to achieve the title *Magister Scientiae*.

After my second year of undergraduate studies I was fortunate enough to be hired for a summer job at the Marine Research Institute of Iceland. I had always favoured pure mathematics over its applied counterpart but this summer was a turning point for me. There my interest and passion for statistics and optimization grew. I realized the power and potential of statistics as a tool to quantify unknown sizes, make accurate and realistic predictions while at the same time taming the uncertainty that follows.

The following summer I continued working on the parallel version of their stock assessment software. This opportunity, for young students to work independently on research and challenging tasks is invaluable and extremely important for the coming generations of scientists and researchers.

I contacted Prof. Gunnar Stefánsson because I wanted to continue my research in a Master's thesis. He welcomed me, introduced me to Prof. Tómas Philip Rúnarsson and has guided me towards a higher academic status.

The path leading to this point has been a valuable experience. I have become well versed in various programming languages, I have learnt to work independently and dive into research head on. I always aimed at pursuing a PhD degree and the work I have done at the University of Iceland has strengthened my interest and allowed me to better see where my interests lie.

Reykjavík May 28, 2014

Guðmundur Einarsson

# Contents

# List of Figures

# List of Tables

# Nomenclature

$\mathbb{X}_{i,j}$ Bernoulli random variable representing value of $(i, j)$ in $C$

$\mathbb{Y}_{c_j}$ Column $j$ in competition matrix $C$, random vector of ones and zeroes

$\mathbb{Y}_{r_i}$ Row $i$ in competition matrix $C$, random vector of ones and zeroes

$C$ Competition matrix

$U(a, b)$ Uniform distribution on the interval $[a, b]$

BFGS Broyden–Fletcher–Goldfarb–Shanno algorithm, quasi Newton optimization method

DE Differential Evolution

EA Evolutionary Algorithm

H& J Hooke & Jeeves optimization method

MOMO Multi-objective-meta-optimization

MRI Marine Research Institute of Iceland

rfa relative fitness assessment

SA Simulated Annealing heuristic

TOCMA Thinned out competition matrix approach

WLC Weighted linear combination

# Acknowledgments

I want to thank first and foremost my main advisor Prof. Gunnar Stefánsson for suggesting this project and introducing me to the world of statistics. He has provided important guidance and helped me in numerous sessions where we have discussed and progressed with the main ideas. Secondly, I would like to thank Prof. Tómas Philip Rúnarsson for pushing me to write a paper with him and Gunnar and providing valuable ideas and critical discussion about various aspects of the thesis.

I would like to thank the Marine Research Institute of Iceland for their generous funding of my thesis. It has allowed me to stay more focused on my work and spend more time on it. They provided me with a summer job to begin with, for which I am ever grateful. I want especially to thank Guðmundur Þórðarson P.hD. for giving me the opportunity to work there in the first place and introducing me to `gadget`, Bjarki Þ. Elvarsson for providing guidance, ideas and inspiration, Björn Ævarr Steinarsson for considering me to work there and finally I want to thank Deputy Director Ólafur S. Ástþórsson and Director Jóhann Sigurjónsson for deciding to fund this project and being helpful and supportive. I want to thank all the other staff at the fisheries advisory section, they play an important role in our society, protecting against overfishing and guarding what the sea has to offer in an intellectual manner.

The professors have all been excellent and I would like to thank them for their hard work and guidance. It is truly amazing that the University of Iceland can provide such quality courses even at the graduate level. I want especially to thank Ragnar Sigurðsson, Birgir Hrafnkelsson, Benedikt S. Magnússon, Rögnvaldur G. Möller, Reynir Axelsson, Eggert Briem, Robert J. Magnus, Jón I. Magnússon, Jón K. Arason, Hjálmtýr Hafsteinsson and of course my advisor Gunnar Stefánsson and my co-advisor Tómas Philip Rúnarsson.

My fellow students have provided support, interest, general suggestions and friendship. I want to thank Ólafur B. Davíðsson, Óli P. Geirsson, Helgi Sigurðarson, Máni M. Viðarsson, Atli Norðmann Sigurðarson, Hjörtur Björnsson and Jóhann S. Björnsson.

I want to thank my family and friends for their love and support. I want especially to thank my soon to be wife Hildur for enduring all the long evenings spent studying and researching, and all the patience she has provided.

# 1. Introduction

## 1.1. Motivation

The work in this thesis was motivated by the tedious optimization procedures that can sometimes occur in the estimation of parameters in large ecological and statistical models. One seeks to minimize the time taken for each optimization method by tuning the method's parameters efficiently. This serves the purpose of speeding up the work process on large models and also to create default optimization parameters for new users. Most of the work for a specific model is related to assessing the results of the optimization. Therefore one wants to make sure that the optimization does in turn return the true optimum. This process of tuning parameters in optimization methods is called *meta-optimization* and is a rich field of research. Parameters that ensure convergence to the true optimum regardless of starting position are desired.

The specific modeling tool that motivated this project is `gadget` (See appendix A), which is the stock assessment software of the Marine Research Institute of Iceland. Three optimization methods are currently implemented in `gadget`, namely Simulated Annealing (Corona et al., 1987; Goffe et al., 1994), Hooke & Jeeves (Hooke et al, 1961) and BFGS (Bertsekas, 1999). The first one is a *heuristic* while the second two are *exact algorithms*.

The user base of `gadget` is rather small so it is desirable to find good parameter values for the optimization methods for medium sized models. This would remove the parameter tuning part for new users and abstracting the optimization. This would further help new users to use the software, since it has a rather steep learning curve. The use of this software tool already requires knowledge from many disciplines, ecology, fishery science, statistics and modeling. Speeding up the learning phase and making the software simpler is important in order to attract new users.

The main concepts of this thesis are speculations and ideas from Prof. Gunnar Stefánsson who has committed countless hours to the development, design and creation of the `gadget` software. Many others have made contributions to `gadget` and it has been researched and used extensively for over twenty years. It is one of the best stock assessment software available today and must be regarded as such and it is

important to use every possibility to spread its usage. More users does not only translate directly to more usage, but also more criticism, new modeling problems and new perspectives on how to solve these problems.

## 1.2. Problem Description

### 1.2.1. Description

The problem is to perform a meta-optimization on the optimization parameters in the `gadget` program. As stated above, `gadget` includes three optimization algorithms. The first is Simulated Annealing which serves the purpose of global optimization, that is to get in the vicinity of the true optimum. Once close the others follow, Hooke & Jeeves is a coordinate based search which progresses closer while BFGS is a quasi-Newton method which approximates the gradient to get as close as possible and can approximate the Hessian of the likelihood function at the optimum point. The latter two are local search algorithms. For these it is assumed that the likelihood function is nearly quadratic around the optimum, which leads BFGS to approach the solution in very few iterations of the algorithm. These three methods are further described in appendix. B.

Usually the parameters of the methods, such as the number of iterations and the *temperature* in SA (Simulated Annealing) are set to extremes which lead to long runs of the algorithms. This of course gives confidence in having found the true optimum after the run, but there is no quantifying reason for the choice of optimization parameters. We therefore desire to find good enough parameters that enable us to arrive at the true optimum regardless of starting values. To ensure this quality of the parameters we need to find hard problems. These hard problems include starting values for the optimization that are less likely to lead to convergence to the true optimum. Examples of such problems are pathological cases such as local optimum points which are hard for the optimization algorithms to escape.

A similar problem has been researched by Hillis (1990), who finds pathological hard cases to tune parameters for the generation of minimal sorting networks. Similar problems have also come up in the tuning of artificial players of various games, (a literature review is provided in section. 1.3). The existence of pathological hard problems may also reveal something else. If there are no local optimum points, then the hard problems are most likely situated close to the boundary of the parameter space, far away from the optimum. But the existence of a local optimum points may give hints of model misspecification (Stefánsson, 2003; Drton et al., 2004). Therefore, the search of hard problems for the meta-optimization may lead to the

finding of local optimization points which in turn may serve as a diagnostics tool for the model building.

We will develop a method that serves this purpose and uses techniques from the framework of evolutionary algorithms. The meta optimization on the methods in `gadget` is very time consuming. Therefore it is important to develop simpler problems that show the capacity of the method to find hard problems and will allow us to assess whether it is possible to diminish the number of function evaluations. This is done in line with the way heuristics are developed for specific problems. These tests show the method's capacity to find other local optima and harder problems in the absence of other optima.

### 1.2.2. Goals

The final goal is to perform the meta-optimization procedure on one distinct `gadget` model and assess the outcome from the procedure. This is a stock assessment model that is similar to the ones used every year in the stock assessment. These models are usually similar in structure each year with new data added annually.

We will start by designing the algorithm (see section 2.4) used for this purpose. This design is described in section 2.4. After that we describe the design and properties of the tests which are used to assess the performance of the algorithm. The results from these tests are presented in chapter 3. Any further modifications before the final tests are presented at chapter 3, prior to the results from the runs with `gadget`.

The goal with these smaller test cases is to show that it is more desirable to reduce the total number of function evaluations and to show the capacity and robustness of the algorithm to solve the problems at hand. This algorithm uses ideas from *coevolution* (see chapter 2) which does **not** give rise to a static objective function evaluation for a given candidate solution. The general method to assess the *score* of a solution in coevolution is called *relative fitness assessment* and a method is explored which shows greater robustness when the number of function evaluations is reduced systematically. Although we don't have static objective function evaluations, we still have a function to measure the score **relative** to the other species.

The overall goal is therefore to show that coevolution can be used efficiently in meta-optimization. Coevolution introduced the notion of *species* (see table 2.1), which are parameter vectors which represent candidate solutions to some problems. Normally there is only interest in one of the species. Here we show possibilities to exploit both species in the coevolution. The terms regarding the coevolution are further explained in chapter 2.

### 1.2.3. Contributions

The contributions to the field of metaheuristics can be summarized in the following list:

(1) A new method to calculate relative fitness assessment, similar to $k$-fold relative fitness assessment is presented in section 2.4.2. The notion of a competition matrix is introduced, which may help others to understand how relative fitness assessment works.

(2) A method to quantify performance of coevolutionary algorithms, called *drift* is presented in section 2.5. Results are presented in chapter 3 which show how this is used to select a parameter for the algorithm in section 2.4.

(3) The usage of the collaborating species as a diagnostics tool for multimodality is discussed and presented in section 3.1.5. This gives use of the collaborating species, but it has normally been used to create pathological anomalies, here it can give a deeper insight on the problems one is trying to solve.

In chapter 4 other variants of the methods are discussed which could lead to future research. There are many different ways possible to tackle the problems at hand and coevolutionary algorithms offer more areas of research which are desired to be explored.

### 1.2.4. Limitations

The obvious limitation of almost all meta-optimization algorithms is that they can take an extremely long time to run. Also since we are working with heuristics, we cannot guarantee the best possible outcome. The method may superficially look like a *black-box* method where one simply clicks the *on-button* and wait for the results, but we have to tailor the algorithm for the job at hand. We have to show that it serves its purpose on examples that have similar characteristics as the larger problems. This is because there are not many convergence results for such algorithms, so the algorithm designer has to rely on effectiveness on smaller problems that portray the same characteristics as the larger one. One thing that can really affect the length of the runs is the dimensionality of the problem. This fact is often called *the curse of dimensionality*.

Another limitation is of course that this algorithm is restricted to certain kind of meta-optimization problems, i.e. we cannot create a black-box method according to the *no free lunch theorems* (Wolpert, 1997). The optimization problems we are

working on have to be bounded, (otherwise the starting points will just wander off towards infinity), and preferably they have to be defined by a continuous function, which has continuous derivatives around the optimum. Those restrictions are mainly posed by the BFGS method, the other ones are more general, yet not as effective on these specific problems.

Yet another issue in competitive coevolution is the possibility of non-transitivity, i.e. cyclic behaviour. This can happen when the competing species are the same, i.e. tuning a strategy model for opposing players in some game, like chess or backgammon. In the case where the species are different, only one of them can find its optimum. This may lead the other species starting to wander off, or start some cyclic behaviour. Therefore it is not simple to create stopping criteria other than simply the maximum number of iterations. One has to assess *a posteriori* the effects of the coevolution and explore the behaviour to deduce what has happened.

Formally, one can not prove directly any convergence results. All results regarding convergence must be shown empirically.

### 1.2.5. Other Applications

The tools used to design the coevolution are taken from the framework of evolutionary algorithms. It should not be to hard to generalize the algorithm further. For example one could try out more variety in the optimization algorithms, so basically we increase the dimensionality by adding more algorithms. One could also add parameters to control how many algorithms are used and in what order. It is possible to do the same for the problems. The problem can be any family of functions parameterized by a parameter vector.

Of course this setting does not have to be restricted to optimization. The evolutionary algorithms were first and foremost created for design. This setting can be applied to design problems, or in fact just about anything that can be represented by a parameter vector and needs tuning against various other problems or scenarios.

When the dimensionality of the problems rises too much, then it is harder to see if cyclic behaviour arises. One has to be able to assess beforehand what kind of problems one may run into, i.e. one has to be able to see which species has the potential to find its optimum.

## 1.3. Literature Review

### 1.3.1. Evolutionary Algorithms and Heuristics

Evolutionary algorithms emerged during the sixties, seventies and eighties (Holland, 1975; Goldberg, 1989), and were algorithms inspired by the process of evolution. Evolutionary algorithms (EAs) is a collective term for various methods, inspired by the biological process of evolution, that emerged in that era. These algorithms were initially intended for design problems but soon people realized their potential as optimization algorithms and heuristics. The most famous evolutionary algorithms used for optimization today are probably *Differential Evolution* (Storn et al., 1995) and the *Covariance-Matrix-Adaptation Evolutionary Strategy* (Hansen et al., 2003).

EAs are really a framework/toolbox/general-scheme to design heuristic methods where one exploits some nature of the problem to do so. This has created some genuinely good optimization algorithms, like the ones named in the previous paragraph. It is hard to explicitly demonstrate that such heuristic methods will converge to the optimal solution, but it can be done in some simple cases, like for the heuristic optimization method *simulated annealing* (See appendix B.1).

The goal of these methods is to search the solution space efficiently to find near-optimal solutions. It is also often the case that it is easy to parallelize these methods. The lack of theory concerning convergence for these methods made them less respectable in the eyes people who worked with exact algorithms. The following phrase is from Glover (1977):

> "Algorithms are conceived in analytic purity in the high citadels of academic research, heuristics are midwifed by expediency in the dark corners of the practitioner's lair... and are accorded lower status".

Fred Glover is the scientist who coined the term *metaheuristic* (Glover, 1986) (See chapter 2). It is not until the nineties that these methods were put on equal footing with exact algorithms. His definition describes a metaheuristic as a framework or a toolbox. The use of such tools had been described by Pólya (1945), a truly innovative addition to the field of heuristics. Simon et al. (1958) describe how the theory of heuristics should be more focused on *intuition, insight* and *learning*. Pólya similarly introduced three principles for creating these methods, namely: *analogy, induction* and the observation of an *auxiliary problem*. So we have to try various ways to exploit the problem at hand in the creation of the method and use the tools that have been made for this purpose. On the other hand, we can try to improve this toolbox, by finding some new tools to add to it. These tools work like modules and can be coupled together in various ways. These principles are what drives research

in the field of metaheuristics.

Sörensen (2013) describes how a plethora of *"novel"* methods has been rising since the beginning of the nineties. There is no unified way of estimating how good heuristic methods are, yet people try to benchmark these methods on a variety of functions. This has created some kind of a *"competition"* in the heuristic community, which is not what science is about. Some of these new methods are created as if prior knowledge and literature is vastly ignored. A good example of this and probably the most notorious of those methods is *Harmony Search* (Geem et al., 2001) (more than 1500 citations). The creator consistently continues to spread his gospel and an improved version of the method even came to existence into 2007 (Mahdavi et al., 2007) (around 700 citations). One can also look at the wikipedia page of Harmony Search to get an idea of what is going on. Sörensen's article is called *Metaheuristics-the metaphor exposed* and it gives a good overview of the situation. Sörensen speculates why this is affecting the field of metaheuristics so badly, but also points out cutting edge research in the field. Although he has some theories about these anomalies, it is important that new researchers and students of this field are made aware of this, especially so that these anomalies don't become the norm!

### 1.3.2. Coevolution as a Metaheuristic

*Coevolution* is a term from evolutionary biology. It symbolises the change in a biological object triggered by the change in another biological object. It can happen cooperatively and competitively, and it can happen at the microscopic-level in DNA as correlated mutations or macroscopic-level where covarying traits between different species change.

Hillis (1990) is the first person to publish an article on the usage of competitive coevolution as a metaheuristic. His paper has at the time this thesis is written, over 1100 citations. He used the competitive nature of the algorithm to create abnormal/pathological cases to deal with, which helped tune another algorithm for generating minimal sorting networks so it would not get stuck at local optima when dealing with such extremities. This setup has inspired others to do the same, when either tuning optimization algorithms or training/tuning game-playing strategies.

When the species are in some sense symmetrical, i.e. representations of opposing players in some games, then it can occur that no strategy dominates all others, i.e. we have non-transitive players. One of the players can always find a strategy to counter what the other player is doing. This can create cyclic behaviour, and has been studied (Samothrakis et al., 2013). Although our problem is transitive in nature, cycling can still occur (De Jong, 2004) and this further emphasizes the importance of the posterior assessment of the behaviour in the coevolution.

### 1.3.3. Multimodal Likelihoods

Warnes et al. (1987) describes the problems that can arise in likelihood estimation of covariance functions of spatial Gaussian processes. The problems presented in the paper show that multimodality can arise, and maximum likelihood estimation may no be the best way of estimation. Further research has been presented by Stefánsson (2003) and Drton et al. (2004). The main results is that lack of data can give rise to multimodality in the likelihood for some specific models and on the other hand, if we have wrongly specified the model we can get asymptotic multimodality in the likelihood, so multimodality may give a hint of model misspecification. An example of a unimodal and a multimodal function can be seen in figure **??**, a 2-dimensional multidmodal function can be seen in figure 2.3.

## 1.4. Structure of the Thesis

The remainder of this thesis is structured as follows. The second chapter goes into greater detail of how the methodology is structured, first the field of EAs is thoroughly introduced in section 2.3, then the structure of the algorithm built for this thesis is broken down in section 2.4. Section 2.5 shows the auxiliary problems created to test the algorithm and show that it can solve similar smaller problems, which have the same possible pathological features as the larger problem. In chapter 3 the results are presented and followed by a discussion, interpreting the results in chapter 4. In chapter 5 possible future work is discussed and the results are assessed further.

# 2. Methodology

This chapter is structured in the following manner. First traditional/classical optimization is discussed in section 2.1, then a brief history and definitions of heuristics and metaheuristics are introduced in section 2.2. After that the framework of evolutionary algorithms is explored and explained in section 2.3, then the design of the coevolutionary algorithm and some pseudocode is explored in section 2.4 and finally the auxiliary problems used in the building and design of the algorithm are explored in section 2.5 and the `gadget`-problem is discussed briefly in section 2.6. A brief summary is provided given in section 2.7.

## 2.1. Optimization

Optimization is the search of a best-quality solution regarding some specific scenario/problem. It may be a simple practical problem such as finding how much coffee you can put in your cup from the coffee-machine that enables you to walk to your desk *without spilling a single drop*!

Generally we are dealing with some problems that can be formalized mathematically. So we are usually trying to minimize or maximize some function. We have some space of candidate solution to our problem, let us call it $S$. It can be a subset of $\{0,1\}^n, \mathbb{Z}^n, \mathbb{R}^n$ or even $\mathbb{C}^n$. The optimization problem is defined by some function $f : S \to \mathbb{R}$. Without loss of generality, we can restrict our problems to minimization problems, (since maximization problems can be solved by minimizing $-f$). So we want to find some $x_F \in S$ such that $f(x_F) \leq f(x)$ for all $x \in S$.

In some cases this can be done by simply solving an equation, but things are usually not that simple. If we cannot resort to solving an equation, the next step is usually to try some *exact algorithms*. These algorithms are deterministic and are often referred to as classical optimization. Some classic methods can be found in Bradie (2006), to take one example we can look at the Newton-method. The Newton method works on functions $f : \mathbb{R} \to \mathbb{R}$ which have continuous first and second order derivatives.

The algorithm iterates successive values with the following iteration formula:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

It is a really strong assumption to know the first and second order derivatives, and things get worse when we generalize the Newton-method to higher dimensions! If we have a function $f : \mathbb{R}^n \to \mathbb{R}$ that we wish to optimize, then the Newton iteration formula is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - H^{-1}_{f(\mathbf{x}_n)} \nabla f(\mathbf{x}_n)$$

where $H^{-1}_{f(\mathbf{x}_n)}$ is the Hessian of $f$ evaluated at $\mathbf{x}_n$ and $\nabla f(\mathbf{x}_n)$ is the gradient of $f$ evaluated at $\mathbf{x}_n$. The Hessian is full of second order partial derivatives. Today there exist symbolic calculation packages that can deal with some of these calculations, but then we are also putting strong assumptions on the function $f$ we are trying to optimize, that the first and second derivatives exist everywhere.

Other methods exist that generalize these methods by approximating the derivatives with secant methods. BFGS is one of them and it is explained further in appendix B.3. Some problems require some different approaches such as *heuristics*.

## 2.2. Heuristics and Metaheuristics

Optimization problems are being solved millions of times everyday on all kinds of systems. In the beginning of the development of such algorithms, implemented for computers, people used *exact algorithms* to solve these problems. Exact algorithms are deterministic in a sense that for a given starting configuration they always return the same result. Another important property of the exact algorithms is that for certain functions it is quite easy to show that they will always converge to the true optimum and one can also derive other properties, such as the speed of convergence and bounds on the error. Some of them can even characterize the types of optimums one finds, such as saddle points or peaks. These algorithms are good enough in most cases of simple optimization.

There is still a problem with these algorithms, they put strong conditions on the objective function. Sometimes the gradient information is needed and even the Hessian, e.g. Newton-method. Another assumption which is usually implied in the usage of these algorithms is the unimodality of the function which is to be optimized. When one applies these methods to multimodal functions, they tend to change into a greedy search, which goes in the direction of the nearest optimum.

During the middle of the last century, scientists realized that these exact algorithms would not always be applicable, the problems they were solving did not always have

these desirable properties. Scientists were creating larger and larger models which required optimization of functions which were not obviously unimodal, and other problems which were almost as a black box with input and output. Imagine you have some model describing the walk of a robot, and the only output for a given test set of parameters is how far the robot can walk. We have little knowledge of how the function works, because there is also some noise in the function evaluation. So the optimization problems scientists were dealing with were not only, not differentiable functions, but also included stochastic components. In the case of the robot, one could also have a restriction on the maximum number of function evaluations. The restriction on the number of function evaluations may also be caused by the length of time it takes to perform the function evaluations. Another such example is in the design of aircraft wings, where the piece which is being designed needs to go through some wind tunnel examination. In these cases approaches like surrogate models have been used to approximate the function with as few function evaluations as possible. Other methods include simulations.

One of the most notorious problem of all of these is the travelling-salesman problem, which can be set up as an integer optimization problem. This problem usually has a unique best solution, but the time it takes to find it grows faster than polynomial-time when the number of knots in the graph increases. This problem, and other similar ones, have required algorithms that can find good approximate solutions fast, and from that work emerged the *heuristic* methods.

What exactly is a *heuristic*? Before the term was used for optimization procedures it meant only: *enabling a person to discover or learn something by themselves.* The same thing almost applies to heuristic optimization methods. The methods don't rely too much on the problem at hand or have too many assumptions, but *learn* the optimal solution. Many heuristic methods are based on *rule of thumb* methods used in searching. One such method is Tabu Search (Glover, 1989), created by Fred Glover. This method somewhat resembles a man looking for water in the desert. He does not want to visit the same place over and over again, so he excludes previously visited regions from his search space and puts them on his taboo-list. Other such rules of thumbs were exploited in optimization methods and people started looking for inspiration from other places than human intuition. So let us give a more concise definition of the word heuristic in the sense of optimization:

**Definition 1.** *A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed.*

This definition is very open, it works more as guiding principles rather than a definition. It is not an easy task to thoroughly define a heuristic. Note that here the classic methods can be more than just the exact algorithms. The heuristics replace

the classic methods, so this refers to the fact that heuristics are often *ad hoc* methods, tailored for specific problems, but still are often applicable to more general problems.

Heuristic methods often incorporate some randomness, so if you run the same method twice with identical starting configurations, there is no guarantee that the results will be identical. Now that we know what a heuristic is, we can look at the definition of a metaheuristic. This term was coined by Glover (1986), who gave the following definition:

**Definition 2.** *A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms. The term is also used to refer to a problem-specific implementation of a heuristic optimization algorithm according to the guidelines expressed in such a framework.*

So a metaheuristic is a framework to build heuristic algorithms and it can also be a specific heuristic created with the framework. Essentially it is a set of consistent ideas, operators and concepts to build heuristics. One could say that metaheuristics are the *skeletons* of heuristic methods. The field of research concerning metaheuristics has vastly increased in the last thirty years or so, and is today one of the richest fields of research in optimization. Most of the literature on metaheuristics is experimental in nature, based on empirical results from computer experiments on the algorithms. So there is a lack of consistent theory regarding many convergence properties and other factors. This could in fact be due to the complexity of the methods and the generality of the framework. In this thesis we are going to work within the evolutionary algorithm framework, to be more precise, coevolutionary algorithms.

## 2.3. Evolution in Optimization

Evolutionary algorithms are special types of heuristics and metaheuristics. They are what is called generic-population based metaheuristics. They are formed from the analogy of evolution and the main aspect borrowed is the *survival of the fittest*. The methods developed from this framework do not need rich domain knowledge, but it can be incorporated. Some technical terms have special synonyms in the theory of evolutionary algorithms, which are summarized in table 2.1. The use of these words for predefined terms may cause confusion, but they are coherent and convenient when discussing these things regarding the ideas of evolution. *Evolutionary algorithms* is a collective term for *genetic programming*, *evolutionary programming*, *evolutionary strategies* and *genetic algorithms*. These classes of methods differ somewhat on what problems they are applied to, but the main idea is the same. These methods have

been used on various optimization problems, including numerical (global) optimization, combinatorial optimization of NP-hard problems, mixed optimization, constrained optimization, multi-objective optimization and optmization in a dynamic or uncertain environment.

*Table 2.1: Synonyms in evolutionary algorithms*

| **EA term** | **Regular term** |
|---|---|
| *Fitness* | Objective/Quality |
| *Fitness landscape* | Objective Function |
| *Fitness Assessment* | Computation of objective function value |
| *Population* | Set of candidate solutions |
| *Individual* | Candidate solution |
| *Generation* | Population produced during a specific iteration |
| *Mutation* | Alter candidate solution independently |
| *Crossover* | Alter candidate solution with other solutions |
| *Chromosome* | Solutions parameter vector |
| *Gene* | Specific parameter in the Chromosome vector |
| *Allele* | Specific value of a gene |
| *Phenotype* | Classification after fitness assessment |

The best way to understand evolutionary algorithms is probably to see some generic pseudocode explained, e.g. algorithm 1 below. As described in algorithm 1, we need an objective function for the input and we output the best individual from the run of the algorithm. First we generate an initial population, which is normally created with some uniform randomness, although here is the place to add some *a priori* information about the distribution of the solutions in the search space. Next there begins a loop where every new population is created, each iteration of the loop is a generation. First the current population is evaluated with its corresponding fitness. Then we select the ones with the highest fitness and generate the next population from their characteristics, using mutation, crossover or other operators that create individuals in a neighbourhood of the existing ones. The creation of new individuals can clearly be visualized in the two dimensional case, as can be seen in figure 2.1.

## 2.3.1. Mutation and Crossover

### Mutation

In the beginning, people only treated the chromosomes as binary vectors, so a typical mutation operator would simply be to switch the value of a random gene. This representation was generalized to cover chromosomes with continuous genes. This is of interest in this thesis, so it will be explored further in the following text.

**Input** : A fitness landscape
**Output**: Best individual

```
Generate initial population P(0);
i ← 0;
while Halting criterion is not met do
    Evaluate fitness of each individual in P(i);
    Select parents based on fitness from P(i);
    Generate offspring with crossover and mutation to form P(i + 1);
    i ← i + 1;
end
```

**Algorithm 1:** A generic EA.

Mutation and crossover are general methods to *tweak* the individual. In fact these operators give us the possibility to create potential new individuals, that do not differ too much from the old one. *Mutation* is seen as an asexual way of creating a new individual. If an individual has a chromosome $\mathbf{v} = (v_1, v_2, .., v_n)$, (where $v_1, v_2, ..., v_n$ are continuous parameters), then a new mutated individual $\mathbf{v}_M$ can be created with the mutation operator. At least one of the parameters has been tweaked, one example of such mutation is the following

$$\mathbf{v}_M = \mathbf{v} + X \cdot \mathbf{e}_i$$

where $i$ is drawn uniformly from $\{1, 2, ..., n\}$ and $X$ can be from any distribution, usually a distribution with density which is symmetric around zero. Here, $\mathbf{e}_i$ is the $i$-th unit vector, so in this case the mutated individual only differs by one gene. There are many variations and studies on empirical results regarding the effectiveness of different mutation operators.

### Crossover

*Crossover* is a way to create a new individual sexually. We mix together information from at least two individuals and create a new one. Let us take a look at a simple crossover operator. We sample two individuals from the population, let us call them $\mathbf{a} = (a_1, a_2, ..., a_n)$ and $\mathbf{b} = (b_1, b_2, ..., b_n)$. Now we create a new individual by picking randomly a parameter from $\mathbf{a}$ or $\mathbf{b}$, let us call this new individual $\mathbf{c}$. So $c_i$ is either $a_i$ or $b_i$, we just draw a Bernoulli random variable to choose which one.

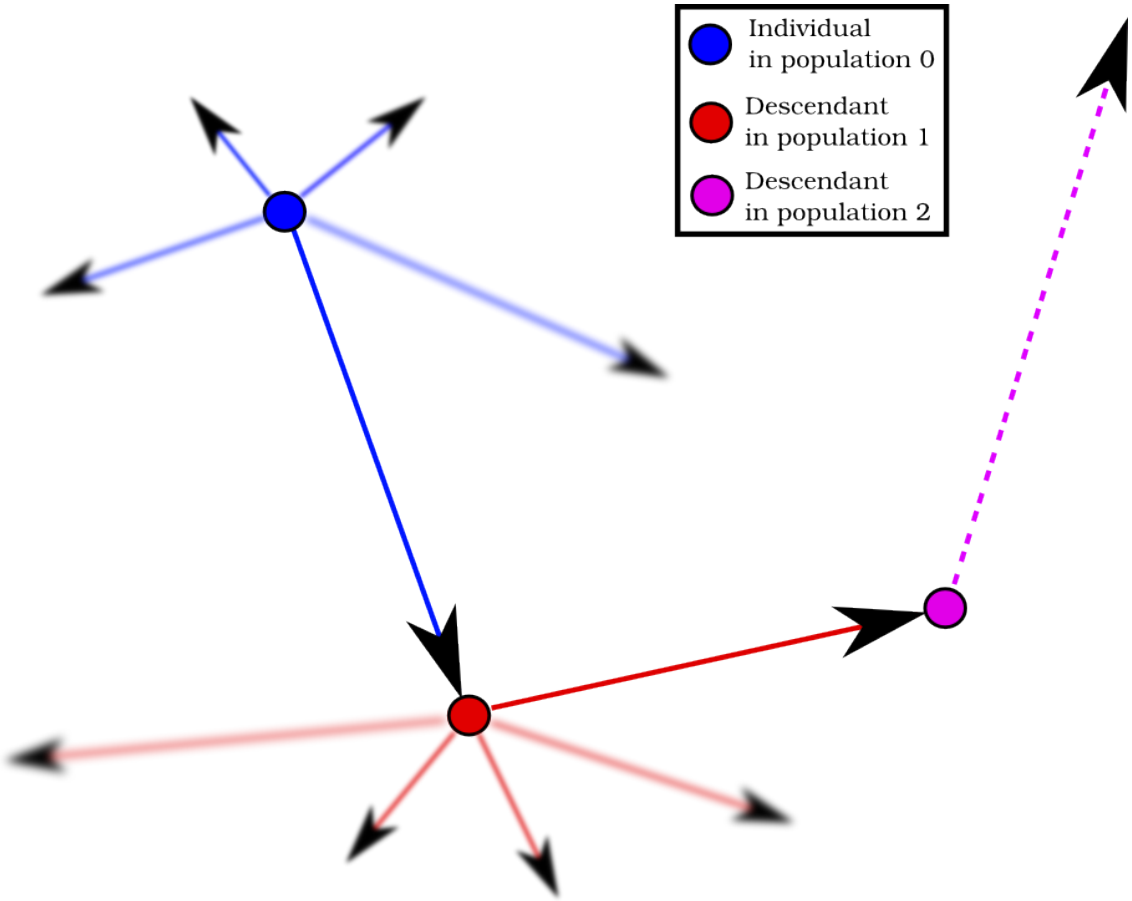


*Figure 2.1: Example of new potential solutions with mutation and/or crossover.*

### Mix of both

We can also have an operation which is a mix of both *crossover* and *mutation*. One such example is of special interest since it will be used later in the thesis to create new individuals in the algorithm designed in section 2.4. This is the operator from *differential evolution* (Storn et al., 1995) which is currently one of the most widely used EA. There are many types used in the general DE implementation, but we are interested in the simplest and most researched one, namely DE/rand/1/bin. The notation is from Storn et al. (1995) and is of the form DE/$x$/$y$/$z$, where $x$ is the vector to be operated on, $y$ is the number of difference vectors used, and $z$ is the crossover scheme and bin is explained below. rand means that we are choosing a random vector. We could also use some elitist strategy, and choose rather those with high fitness, but that can make the algorithm more greedy. This version of differential evolution is characterized by three parameters seen in table 2.2.

*Table 2.2: Parameters in differential evolution*

| **Parameter** | **Purpose** |
|---|---|
| *NP* | Number of individuals in each population |
| *CR* | Cross-over ratio |
| *F* | Multiplicative constant with the differential variation vector |

The procedure to create new individuals is then the following:

Let us say we have a population of NP individuals and we are working with individual $\mathbf{v}_i$ and we want to create a potential descendent. Now we sample three distinct individuals, different from $\mathbf{v}_i$, from the population, let us call them $\mathbf{v}_{r_1}$, $\mathbf{v}_{r_2}$ and $\mathbf{v}_{r_3}$. Now we create a new mutated candidate solution, let us call it $\mathbf{m}$

$$\mathbf{m} = \mathbf{v}_{r_1} + F(\mathbf{v}_{r_2} - \mathbf{v}_{r_3})$$

here $F \in [0, 2]$, (it can also be a random variable in other variations). Now we crossover this chromosome with the chromosome $\mathbf{v}_i$ to form $\mathbf{n}$, the new vector. For each gene we sample one uniformly distributed random variable $r_j \sim U(0, 1)$ and if $r_j \leq \text{CR}$ we choose $m_j$, else we choose $v_{i,j}$ as $n_j$. We also sample one index $R$ from $\{1, 2, ..., \text{NP}\}$ and assign $n_R$ as $m_R$, to ensure that at least one gene changes. Now we evaluate the fitness of this new individual, it replaces the parent if it is better, otherwise we keep the parent in the new generation and discard the offspring.

Values of $F$ far from zero make the algorithm more *explorative* while values closer to zero make it more *greedy*. If the values of the genes are a mix of integers and real numbers, we can choose $F$ as a vector $\mathbf{F}$, where $F_i$ is 1 if gene $i$ is an integer and some pre-specified value otherwise.

The vector $(\mathbf{v}_{r_2} - \mathbf{v}_{r_3})$ is called the differential variation vector. All the combination of other individuals in the population create a set of potential new individuals, but we randomly pick one. The set of all such potential individuals surround the area of the current individual, this can be seen in figure 2.1, where the discarded individuals are slightly blurred.

### 2.3.2. Coevolution

Now we have been thoroughly introduced to the ideas of evolution in optimization and can therefore examine coevolution. In the schemes introduced above, we have only been dealing with a population of one species. We can now extend the ideas to more than one species to get *coevolution*. Of course this is also inspired by ideas from nature, but there we can have both *competitive* and *cooperative* coevolution. The most significant thing that changes now is probably the fact that one doesn't have any stable fitness landscape. The fitness is dependant on the other species, so one only has a relative way of measuring fitness, i.e. a *relative fitness measure*. The relative fitness measure used in the algorithm in this thesis is introduced in section 2.4. Having two species interact increases the complexity of the algorithms dramatically and can cause many problems as described in section 1.3.2. Here we are interested in only two species, so each individual in one species, has to be evaluated against various other ones in the competing species to get a relative fitness measure. This increases the time complexity drastically if everyone competes against everyone. Therefore we need to try to minimize the number of these evaluations. That is the purpose of the auxiliary problems in section 2.5.

In this thesis we are interested in competitive coevolution and the scheme has been named *predator-prey-competitive-coevolution-scheme*. Here the predator chromosome consists of parameters for optimization methods, while the prey chromosome are starting points for the optimization. The prey can in general be any parameters controlling the shape of the function being optimized. So we can look at the problem as if we are tuning optimization algorithms on a parameterized family of functions.

## 2.4. Algorithm Design

The algorithm itself is structured as shown in algorithm 2. There we introduce some functions which are further explained in later sections. The algorithm needs some data to work on, first we need some parameters for the genetic operators, which are identical to the one in differential evolution, see section 2.3.1. Then we define the maximum number of generations $G$. There is no other obvious way to

create a stopping criterion, so *a posteriori* assessment is needed to see how the coevolution went. This is a *constrained* metaheuristic, so we need some bounds on the genes in the chromosomes. Next we need the *score function* which is used for the *relative fitness assessment*, (see section 2.3.2), but some options for these are explored in section 2.4.2. Now we need some predators and prey, some examples are explored in section 2.5 concerning the auxiliary problems, but ultimately we will be using minimization problems in `gadget`, (see appendix A), and the optimization algorithms described in appendix B. Implementations of the functions in algorithm 2 in R can be found in appendix D.

**Algorithm** `Coevolution`
- **Data**: Parameters for the genetic operators;
- Maximum number of generations $G$;
- Boundaries for the gene values;
- Score function;
- Parameterized family of functions (Prey) ;
- Parameter vector defining switches for optimization methods (Predator)
- **Result**: Final populations for both species;
- A matrix representing statistics from each generation;
- Predator = `GenerateInitialPopulation(`*pred*`)`;
- Prey = `GenerateInitialPopulation(`*prey*`)`;
- Score = `CalculateScore(`*Predator,Prey*`)`;
- **for** *k from 1 to G* **do**
  - [NewPredator NewPrey] = `GenerateNewPopulation(`*Predator,Prey*`)`;
  - Score = `CalculateScore(`*Predator,Prey,NewPredator,NewPrey*`)`;
  - [Predator Prey] = `UpdatePopulation(`*Score*`)`;
  - Stats = `Combine(`*Stats,* `CollectStatistics(`*Score,...*`))`;
- **end**
- **return** *Predator,Prey,Stats*;

**Algorithm 2:** The coevolution scheme.

We can summarize the parameters needed for the algorithm in table 2.3. We can see that the parameters are almost identical to those needed in differential evolution, except we have the parameter $p$ which is explained in section 2.4.2 and $G$, which is the number of iterations.

Now there are three functions of interest introduced in the pseudocode, namely `GenerateNewPopulation`, `CalculateScore` and `UpdatePopulation`. These are explored in sections 2.4.1, 2.4.2 and 2.4.3 respectively.

*Table 2.3: Parameters in predator-prey-coevolution-scheme*

| Parameter | Purpose |
|---|---|
| $N$ | Number of individuals in each species |
| $CR$ | Cross-over ratio |
| $F_{\text{vec}}$ | Vector described at end of section 2.3.1 |
| $p$ | Integer parameter controlling number of fitness assessments |
| $G$ | Number of generations |

### 2.4.1. Generation of Populations

First we need to generate the initial population. That is achieved with the function `GenerateInitialPopulation`. This is simply uniform sampling within the predefined boundaries of the genes, note that in this version of the algorithm we have equal number of predator and prey individuals, this can be generalized to different numbers in each species. Here we call the number of individual in each species $N$, so here we have a total of $2N$ individuals in each generation. This function can be changed to guide the search towards prosperous regions in the beginning, although it is not advised, since when the dimension of the problem becomes large, it is almost impossible to have an idea of the structure of the problem.

Now in every successive iteration we need to create new individuals from an already existing population. This is achieved with the function `GenerateNewPopulation`, which uses the method described in section 2.3.1. Notice that $F_{\text{vec}}$ which appears in table 2.3 is a vector of the same length as the species chromosome and takes the value 1 for integer genes/parameters and $F$ for real genes/parameters. It is a way to deal with crossover and mutation for the integer valued genes.

Notice that after we have created the offspring, there are $4N$ individuals in the total population, now we need to select the surviving $2N$.

### 2.4.2. Competition and Relative Fitness Assessment

At this point in the iterations we have created the offspring, so we need to evaluate them with the older individuals. As mentioned several times above, we do not have a direct objective function. We can only estimate the performance of an individual relative to the all or some individuals in the other species, so we have some function to serve that purpose. That function can be seen as an objective function, but it is dependent on the individuals in the opposing species. There exist several ways to perform relative fitness assessment. One is $k$-fold relative fitness assessment (rfa). In the $k$-fold rfa we do the following for every individual:

(1) Sample $k$ individuals from the competing species.

(2) Estimate the fitness of our individual against these $k$.

(3) Collect the scores to create some final score, e.g. the mean or the sum.

This is done for all individuals, so if there are $N$ individuals in each species, we have $4Nk$ fitness assessments, (there are $2N$ individuals of each species at this instance, offspring and parents), or maybe fewer. If we first sample for one species, and than use that information partially for the other species, we have at minimum $2Nk$ evaluations.

The method used in this thesis is very similar to $k$-fold rfa, but what is fundamentally different, is that we don't have a fixed number of evaluations for each individual and our representation of the evaluations may be easier to interpret and generalize. It has been dubbed *thinned-out-competition-matrix-approach* (TOCMA). This is to construct the relations between the predator and prey, to see who goes against who. We construct an $2N \times 2N$ competition matrix $C$ as described in algorithm 3. An implementation of the function `GenCompMat` in R can be found in appendix D.

**Function** `GenCompMat(`$p$,$N$`)`
- **Data**: Integer parameters $p$ and $N$
- **Result**: Sampled competition matrix $C$
- Initialize $N \times N$ matrix $C$ with all zero entries;
- $C =$ `SampRows(`$C$,$p$,$N$`)`;
- $C =$ `SampRows(`$C^T$,$p$,$N$`)`;
- **return** $C$;

**Procedure** `SampRows(`$C$,$p$,$N$`)`
- **for** $i$ *from 1 to* $N$ **do**
  - RndInd = $p$ numbers sampled from $S = \{1, 2, ..., N\}$;
  - **for** $k$ *from 1 to* $N$ **do**
    - **if** $k \in RndInd$ **then**
      - $C_{ij} = 1$;
    - **end**
  - **end**
- **end**
- **return** $C$;

**Algorithm 3:** Generating the competition matrix $C$.

Let us go over this in more detail. For each row and each column of the matrix $C$ we sample $p$ distinct indices from $\{1, 2, ..., N\}$, (where $N$ is the input into the algorithm). Let us say that we are doing this for a given row $i$, then we have $p$ indices, namely $j_1, j_2, ..., j_p$ and we set $C_{ij} = 1$ if $j \in \{j_1, j_2, ..., j_p\}$. All the values are initialized as zero. So now the number of 1s in row $i$ is $p$, but more could be added when we sample for the columns!

Let $\mathbb{Y}_{r_i}$ be the random $i$-th row created of ones and zeroes in $C$ and $\mathbb{Y}_{c_j}$ the random $j$-th column. Denote by $\mathbb{Y}_{r_i(j)}$ the $j$-th element of $\mathbb{Y}_{r_i}$ and $\mathbb{Y}_{c_j(i)}$ the $i$-th element of $\mathbb{Y}_{c_j}$. So for a given pair of indices $(i, j)$ we have that $\mathbb{Y}_{r_i(j)}$ and $\mathbb{Y}_{c_j(i)}$ are independent Bernoulli random variables with the probability $\frac{p}{N}$ of being one.

Now we would like to interpret each slot in the matrix as a discrete random variable $\mathbb{X}_{i,j}$. This is a Bernoulli random variable, and we can calculate the probability of it taking the value one. We have that $P(\mathbb{X}_{i,j} = 1) = P(\max\{\mathbb{Y}_{r_i(j)}, \mathbb{Y}_{c_j(i)}\} = 1)$, and our calculations yield

$$P(\mathbb{X}_{i,j} = 1) = \frac{p}{N}\left(2 - \frac{p}{N}\right)$$

Now we have the probability, so we can calculate the number of expected ones in the matrix $C$, which again yields us with the expected number of fitness evaluations in each iteration. We have:

$$\mathrm{E}\left[\sum_{i=1}^{N}\sum_{j=1}^{N}\mathbb{X}_{i,j}\right] = \sum_{i=1}^{N}\sum_{j=1}^{N}P(\mathbb{X}_{i,j} = 1) = Np\left(2 - \frac{p}{N}\right)$$

For low values of $p$ this is significantly better than $N^2$, exhaustively making everyone compete. We will see the effectiveness of varying $p$ on some of the auxiliary problems in chapter 3, and that it is well feasible to have values of $p$ as low as 2 or 3 in these cases.

### Parallelization

The fitness evaluation representing the entries in $C$ are completely independent, so this is the perfect place to parallelize the algorithm. These calculations are implemented as a double `for`-loop, which is easily parallelized in R, as shown in appendix C. This is one of the reasons why population based optimization algorithms have become more popular. It is usually very simple to parallelize them.

The parallelization scales almost perfectly when the fitness evaluations are time consuming. If they are fast, then the communication overhead may slow the algorithm down. One optimization call in `gadget` takes a minimum of 2 seconds, so in this the communication overhead becomes negligible.

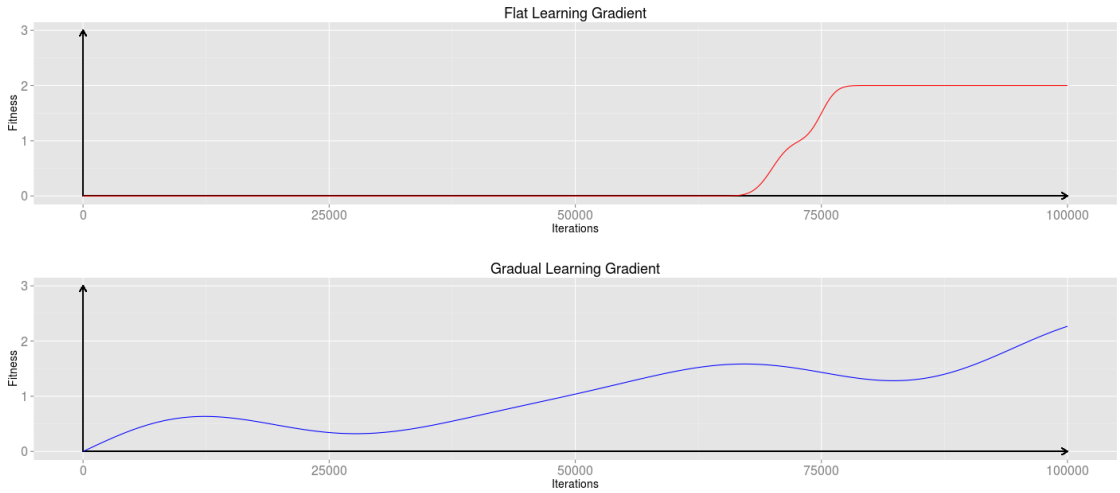


*Figure 2.2: Example of a bad learning gradient, (needle in a haystack problem), and a good progressive one.*

### Why bother?

One could ask now, why all the hassle of having evolving prey? Why not just keep it static as the hardest problem? Luke (2009) explains it well. He creates the analogy of the training poker players. Let's say that you are interested in training poker players. You create some *guru*, (like the hardest problem), which is an extremely strong player. Now you make your new players play against him multiple times in each iteration. The guru may always win, until suddenly the players figure something out and beat him. This may be after a very long time, so we get a very bad *learning gradient*, the fitness value suddenly rises fast at some point after many iterations. This is sometimes called *a needle in a haystack problem*. But there is no way of differentiating between players which are competing against the guru, they lose all the time, so we don't know which players to *breed* and create offspring from for the coming iterations, it's basically a random search for a needle in a haystack.

Now if we stop looking at the guru and let the players play against each other we can get a much better measure of who of them are improving, because we have a relative measure with respect to the other players. Now they can better select which strategies to evolve from and we get a much smoother learning gradient. Examples of learning gradients can be seen in figure 2.2.

### What do we need to calculate?

In section 2.4.2 above, we created the competition matrix, but we need to perform some calculations for each *one* entry in the matrix $C$. The value computed is of course dependent on our goal. We would like to robustify the optimization methods in `gadget`, such that for a given starting point, we always find the global optimum, but we would also like to minimize the time taken for each of these optimization calls. This calls out for *multi-objective* meta-optimization (MOMO). The time taken for an optimization run is directly proportional to the number of function-calls performed by each of the optimizing algorithms, we can therefore use that as a *proxy* instead of time. So we have two goals:

(1) Robustify the algorithms in `gadget` so that they always find the global optimum.

(2) Minimize the number of function calls.

Recent literature includes many ways to deal with MOMO, especially methods concerning *pareto-optimality*, where one finds a *pareto-front* and can select solutions based on what trades of the many objectives are most important. Some speculations about the *pareto-front* can be found in chapter 8 of Deb (2001). Calculating the *pareto-front* for this problem may not be computationally feasible in reasonable time, since the optimization calls in `gadget` can be very tedious and we are simultaneously trying to robustify against hard starting points. Let us take a look at a simpler method.

### Weighted Linear Combinations

We create a weighted linear combination (WLC) of the objectives. Let us call $f_1$ the lowest function value from the optimization in `gadget` and $f_2$ the number of function evaluations used to achieve that. Then we calculate the score behind each one entry in the competition matrix $C$ as:

$$f = w_1 f_1 + w_2 f_2$$

The weights $w_1$ and $w_2$ are used to normalize the objectives. We scale the values such that $w_2 f_2$ is an order of magnitude lower than $w_1 f_1$, in order to ensure convergence. After the methods are converging, then the improvement is in the reduction of function evaluations, but we penalize significantly if the convergence is not achieved. This method gives a single solution on the *pareto-front*, and we can bias it with the weights regarding what we would like to achieve.

### 2.4.3. Updating the Population

Once we have collected the scores for each individual against all the others sampled, we need to aggregate the scores somehow to create a single score for each individual. This is done in the function `UpdatePopulation` from algorithm 2. A given individual $i$ has $n_{(i)}$ number of fitness evaluations $f_1, f_2, ..., f_{n(i)}$. Now we calculate the mean of these values, let us call it $S_{(i)}$, and the standard deviation $\sigma_{(i)}$. If $i$ is a predator, then we are trying to minimize the score, a new individual's only possibility to enter the population is to beat the parent, so we accept this new predator and discard the parent if:

$$S_{(i)\text{child}} + \frac{\sigma_{(i)\text{child}}}{\sqrt{n_{(i)\text{child}}}} < S_{\text{parent}}$$

else we discard the child. Indices have been added to recognise whether a parameter comes from the parent or the offspring. For the prey we have a similar criterion, namely

$$S_{(i)\text{child}} - \frac{\sigma_{(i)\text{child}}}{\sqrt{n_{(i)\text{child}}}} > S_{\text{parent}}$$

so we choose individuals with higher scores, in attempt to create more difficult problems to robustify the algorithms. We need the standard deviation to cap the scores. This prevents the algorithm from chasing some noise and creates an assurance that this new individual is truly better relative to the current population.

This way of only allowing children to replace the parents makes the algorithm more explorative, instead of choosing the $N$ best from the current population of parents and offspring, which is an elitist strategy. It is not as simple as choosing the $N$ best, since we need to take some variation/noise into account.

## 2.5. Structure of Auxiliary Problems

Two smaller auxiliary problems were created to test the performance of the algorithm. More problems were investigated, but these were of special interest, since they capture the qualities of the larger problem. These problems are similar in nature to the predator/prey setup with starting points for `gadget` and its optimization methods. Here we have replaced the likelihood function in `gadget` by another function and we restrict ourselves to one or two optimization algorithms. This is done to be able to perform multiple tests in a feasible time, and show some empirical results. The functions are explained in detail below as well as the idea behind them. The optimization methods used are simulated annealing and BFGS.

There are two main problems that we would like the prey to find if they exist, i.e. local minimums and valleys that are hard for simulated annealing. The auxiliary

problems have exactly that. We use them to see whether changing the parameter $p$(see table 2.3) has a significant effect on the *drift* towards these regions. Let us define *drift*.

**Definition 3.** *For a population-based metaheuristic and some specific prosperous region $P$ in the search space, which is known a priori to include the optimum for this population, the drift is the expected number of iterations it takes for all the population to be inside this region $P$.*

Note that a *prosperous region* is an analogy for a neighbourhood of the optimum. We shall use empirical results to evaluate the drift on the auxiliary problems. Let us see the functions now:

## 2.5.1. Multiple Optima Function

This is a function constructed to show the capacity of the algorithm to find local optima that is hard for the algorithms in `gadget` to get out of. The function is the following:

$$f_{\text{MultMod}}(x,y) = \frac{x^2+y^2}{100} - 10e^{-(x-3)^2-(y-3)^2} - 9e^{-(x+3)^2-(y-3)^2} - 3e^{-(x+3)^2-(y+3)^2} - 8e^{-(x-3)^2-(y+3)^2}$$

The minima are summarized in table 2.4.

*Table 2.4: Position and values of minima in function $f_{MultMod}$*

| **Position** | **Value** |
|---|---|
| $(x,y) = (3,3)$ | $-9.82$ |
| $(x,y) = (-3,3)$ | $-8.82$ |
| $(x,y) = (-3,-3)$ | $-2.82$ |
| $(x,y) = (3,-3)$ | $-7.82$ |

The starting prey are uniformly scattered starting position on the square $[-5,5] \times [-5,5]$ and we start off by looking at the predators as only simulated annealing parameters with limited number of iterations (see table 3.1). We should expect that the prey drifts towards the highest minimum while the predator tries to increase the number of function evaluations until it reaches the upper bound or manages to find the lowest minimum. The predator is also expected to increase the temperature, which is proportional to the scale in the *Gaussian* kernel used to generate new points so the search becomes more random and more likely to potentially end up in the other lower minima. This version of the simulated annealing method is the

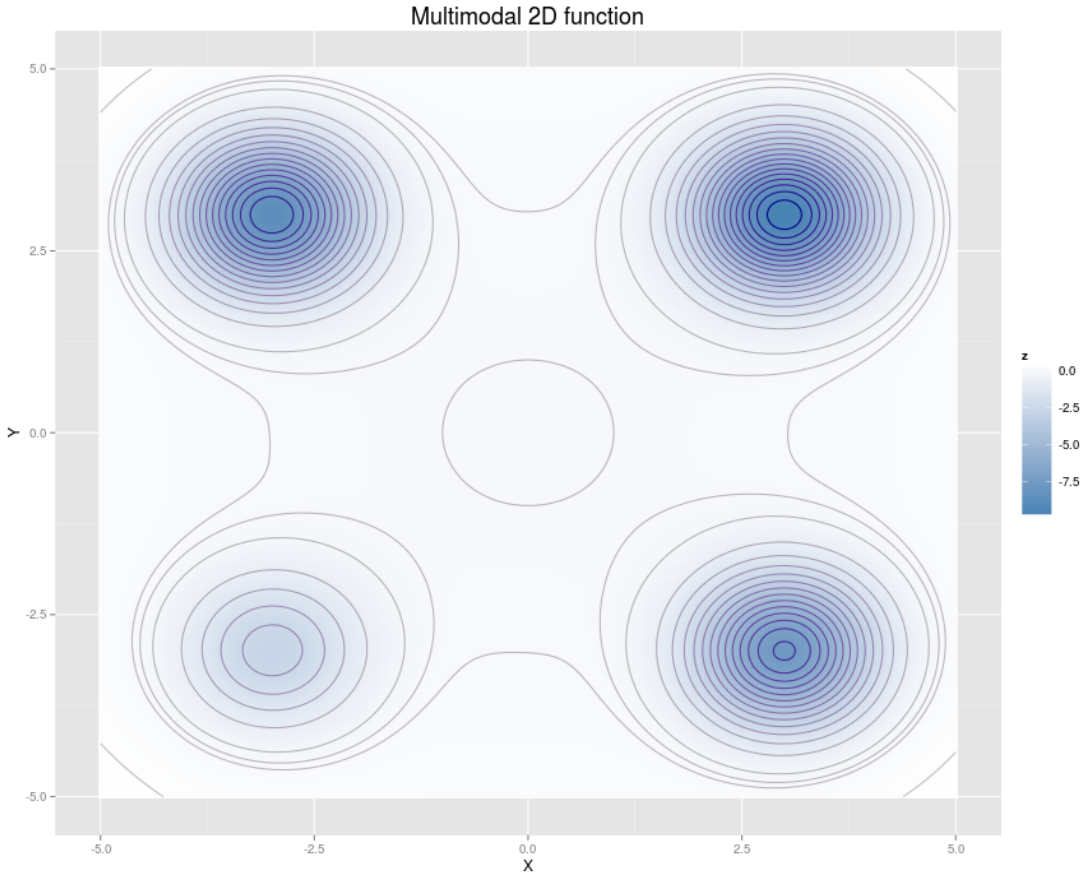


*Figure 2.3: Multimodal 2-D function for experiments on drift, contour plot*

one presented by Bélisle (1990) and implemented in the `optim` function in R. The main idea with this problem is to explore the effect of varying the parameter $p$ in the coevolution scheme to see how that affects the drift. We define the rectangle $[-5, 0] \times [-5, 0]$ as the prosperous region $P$ because it includes the highest local minimum.

For just any problem it is impossible a priori to find the region $P$. That is why we had to specifically construct this problem, so it would be easy to define some $P$ that would obviously be the prosperous region.

In the second experiment with this function we add BFGS to follow simulated annealing and look at some empirical results. A contour plot of the function can be seen in figure 2.3 and a surface plot in figure 2.4.

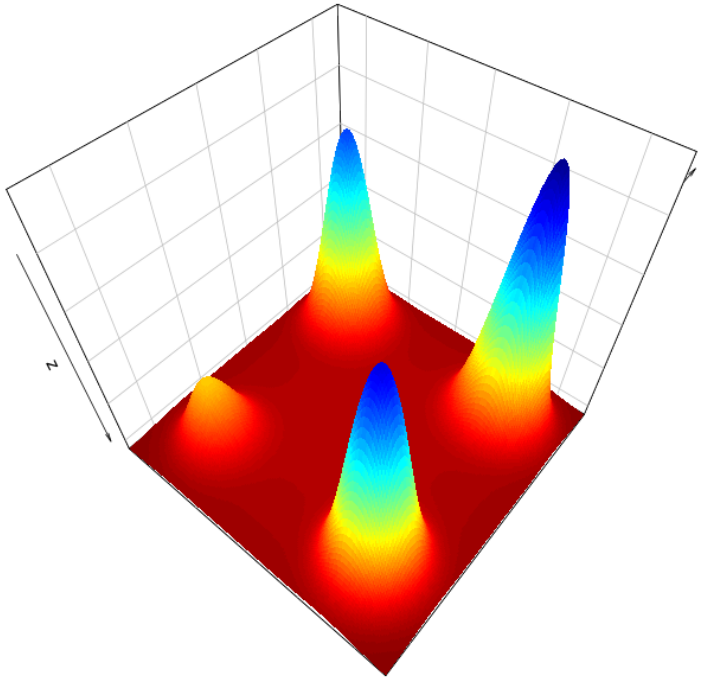


*Figure 2.4: Multimodal 2-D function for experiments on drift, surface plot seen from below*

### 2.5.2. The Rosenbrock Functions

The previous section describes a problem which shows the capacity of the algorithm to find non-global optima, but there are other problems that are hard for some heuristics, especially simulated annealing, i.e. functions which include curved valleys. The most notorious of those functions is the *Rosenbrock-function*, which has been heavily used in the literature for demonstrations, as a hard problem for non-gradient-based methods, whereas some gradient based methods work well on it. This function is usually especially hard for heuristics, but some non-gradient-based methods have been created to tackle these sort of problems, namely *adaptive coordinate descent* which is an evolutionary algorithm and the most recent of these methods (Schoenauer et al., 2011), Rosenbrock's method (Rosenbrock, 1960) is probably the oldest one, but several extension of it exist.

The function has also been named as the *banana-function* because of its curved shape. One reason for its popularity is how simple it is to code and generalize it to higher dimensions. The 2-dimensional version is the following:

$$f_{\text{Ros}}(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

It has a unique global minimum at $(1, 1)$ and we will explore it in the rectangle domain $[-5, 5] \times [-5, 5]$. We expect the prey to "*crawl*" up the valley and find the hardest problems around the point $(-4, 4)$, so here we define the prosperous region as $P_{\text{Ros}} = [-5, 0] \times [-5, 5]$. It's the "left side" of the domain. We use a similar setup to the one in the last problem. The prey starts as uniformly scattered points in the domain. The bounds on the optmization parameters are summarized in tables in chapter 3. A contour plot of the function can be seen in figure 2.5 and a surface plot can be seen in figure 2.6.

### 2.5.3. Summary on auxiliary problems

Let us summarize the tests in table 2.5. These problems will be explored with respect to drift and variation in the parameter $p$ from the coevolution scheme. Some results concerning this work with the simulated annealing algorithm has been submitted for publication in a conference journal. We extend it here by also adding BFGS and show more empirical results. The cases will be called case 1, case 2, case 3 and case 4 according to their order in table 2.5.

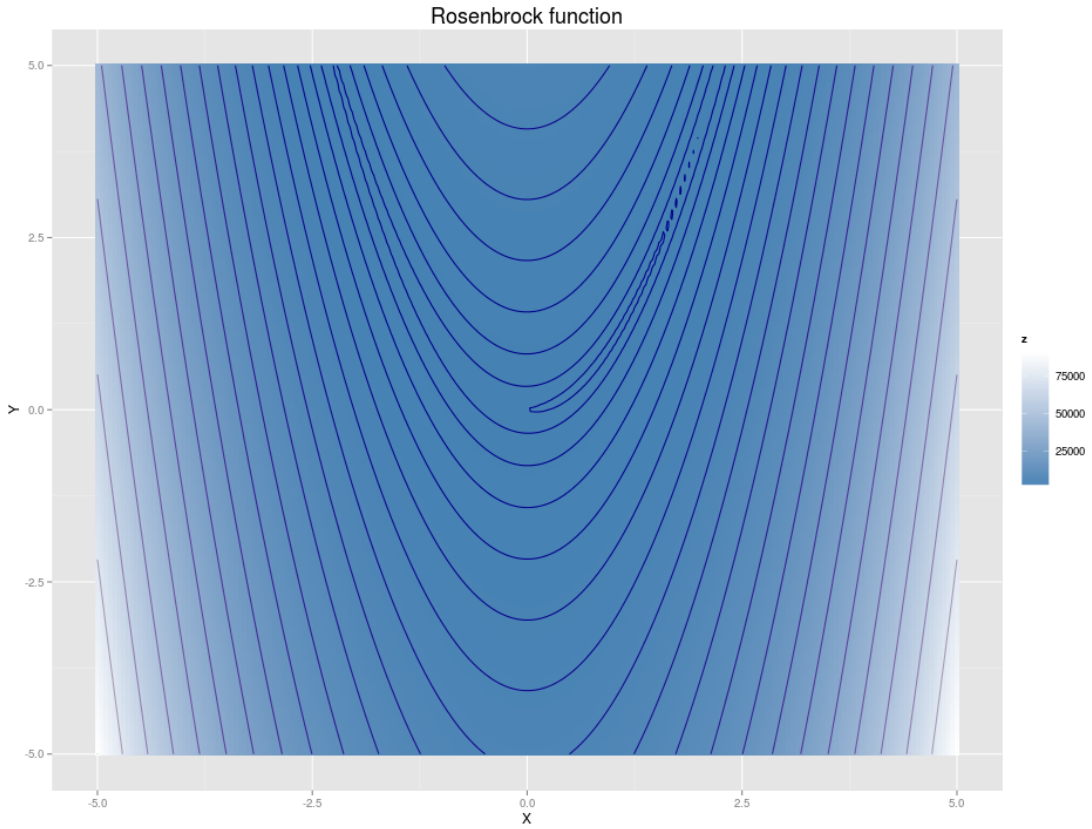


*Figure 2.5: Rosenbrock function for experiments on drift, contour plot*

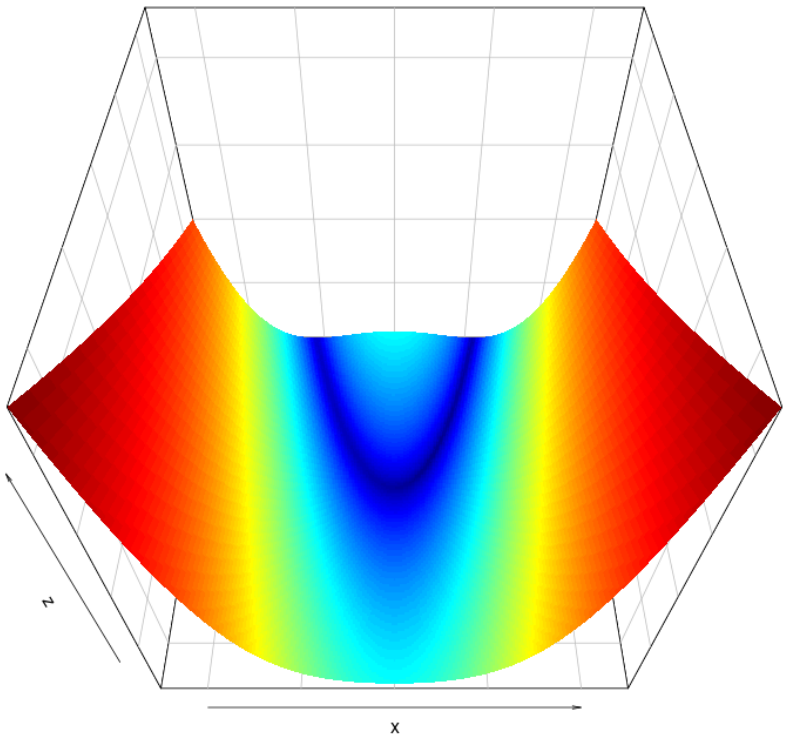


*Figure 2.6: Rosenbrock function for experiments on drift, surface plot*

*Table 2.5: Auxiliary problems, explored with respect to drift and coevolution*

| **Function (Prey)** | **Optimization methods (Predator)** |
|---|---|
| $f_{\text{MultMod}}$ | SA (See subsection 3.1.1) |
| $f_{\text{MultMod}}$ | SA & BFGS (See subsection 3.1.2) |
| $f_{\text{Ros}}$ | SA (See subsection 3.1.3) |
| $f_{\text{Ros}}$ | SA & BFGS (See subsection 3.1.4) |

## 2.6. Gadget Problems

The characteristics of the problems in `gadget` are similar to the auxiliary problems. One may have numerous local optima and there are potential valleys similar to the ones in the Rosenbrock function. The performance of the algorithm will be shown on a problem which is called the *haddock example*; a simple model with simulated data, created for new users to test `gadget` and learn about the various input and output files used. The Prey chromosome has a total of 38 genes, which are the starting parameters in the optimization. The predator chromosome consists of a total of 21 parameters, 10 for SA, 4 for H & J and 7 for BFGS. The prey is not really of interest here, we seek to show that the algorithms can be robustified. The parameters in the optimization methods are explained in appendix B.

## 2.7. Summary

We began by looking at some explanations of basic terms concerning optimization, then we moved onto a historical note on heuristics and metaheuristics. Then evolution in optimization was introduced and some basic terms explained. Note that this is the section where the operators used in the algorithm are explained, the once used in DE. After that the whole coevolutionary algorithm is explained in detail, with some discussion on why evolution is important for this case, i.e how one should try to avoid learning gradients that represent needle in a haystack problems. In the end the auxiliary problems are introduced and explanation is provided for the selection of these problems.

Note that the preys are emphasized more than the predators for the auxiliary problems, because it is easier to explore and visualize them. They are also more obvious to explore with respect to drift, i.e. the collective attraction to a prosperous region. The predator consists of three or four parameters, namely the number of iterations, temperature and time-steps between temperature change for SA and only the maximum number of iterations for BFGS.

# 3. Results

This chapter is partitioned into two sections. Firstly we show the results for the auxiliary problems, which serve the purpose of demonstrating the effectiveness of the algorithm empirically. We are especially interested in seeing the effect of varying $p$. There are 4 cases explored, summarized in table 2.5. Then a short summary is presented on the results and a brief discussion on the potential of the algorithm as a diagnostics tool for multimodality.

Secondly, we look at the results from the algorithm applied to a `gadget`-model and some minor adjustments that had to be made for the sake of dimensionality. These adjustment are further elaborated in section 3.2.

Note that when the predators consist of multiple optimization algorithms, then they are always run in the same order. The outcome from the one before is used as a starting point for the next one.

## 3.1. Results for Auxiliary Problems

The auxiliary problems results are all presented with the following plots:

(1) **Prey *snap-shot*-plots**. These are plots which show the position of the prey individuals at a given generation. These plots are presented mainly for the reader to get a better feel for what is going on. See for example figure 3.1

(2) **History plots**. These plots show the min, max and mean rfa scores and their standard deviation through the generations. See for example figure 3.2.

(3) **Coevolution-plots**. These plots show the evolution of the parameters in the optimization methods, they show min, max and mean values, along with the standard deviations. See for example figure 3.3.

(4) **Varying $p$-plots**. These plots show the effect of varying $p$ with respect to drift as discussed in section 2.5. See for example figure 3.4.

There are tables (e.g. table 3.1) that summarize any bounds on optimization parameters and which parameters are being allowed to evolve. A short discussion on the results can be found prior to the corresponding graphs of each subsection.

The versions of the optimization methods used here are the ones implemented in the `optim` function in R. The BFGS version used is the one published in 1970 simultaneously by Broyden, Fletcher, Goldfarb and Shanno.

### 3.1.1. Function $f_{\mathsf{MultMod}}$ with SA (Case 1)

The plots and results are in accordance with the order in the list presented in section 3.1. The following sections include plots and a short discussion on the results. The parameters tuned in SA are listed with bounds in table 3.1

*Table 3.1: Parameters for SA in Case 1*

| Parameter | Lower bound | Upper bound |
|---|---|---|
| `maxit` | 0 | 1000 |
| `temp` | 0 | 1000 |
| `tmax` | 1 | 100 |

#### Snap-shot-plots (Case 1)

Here we look at a snap-shot of 500 generations where $p = 3$ in the coevolution and $F = 1.5$ for continuous parameters and $CR = 0.78$. The number of individuals in each species is $N = 10$. The results are presented in figure 3.1.

#### History-plots (Case 1)

These plots show some summary function on the outcome from the species individuals in each generation. The summary functions are min, mean and max, which are used on the relative fitness assessment score and its standard deviation.

We can notice from figure 3.2 that the learning gradient is rather smooth in the beginning and then flattens, a loess curve has been fitted to better see what is going on. The hardest problems become progressively harder, but the algorithms become better, and in the end counter the hardest problems, we can see in figure 3.1 that in generation 500 some of the prey has strayed away from the prosperous region, the

predator has won and the prey has a *weakened sense* for what is the best place to stay at. That is because most of the prey are getting similar scores relative to any algorithm in that generation.

### Coevolution-plots (Case 1)

The plots in figure 3.3 are sorted in columns by parameter for the optimization. Each parameter is summarized in one column of the plots. Each dot represents the outcome of this summary function applied to the individuals of one generation.

The most interesting fact from these pictures is probably that the temperature plummets down to one when SA starts performing better on this function. The scale in the Gaussian-kernel is proportional to the step size, so smaller step sizes are more likely to work better on this function. We can also clearly see some collective behaviour for the `tmax` parameter. The standard deviation is dropping and it seems as if the parameter is settling on some value around 70-80.

### Varying-$p$-plots (Case 1)

The purpose of these plots is to demonstrate empirically the effect of changing the $p$ parameter in the coevolution scheme (see table 2.3). These results show that *drift* is dramatically affected by changes in $p$. These results also indicate that it is better to have lower values of $p$ to minimize the number of rfa evaluations, preferably $p = 3$ in this case. The setup here is a bit different to the cases above. The number of iterations for SA is limited to 100 because we sample the drift for each $p$ 100 times and we are mainly interested in the movement of the prey.

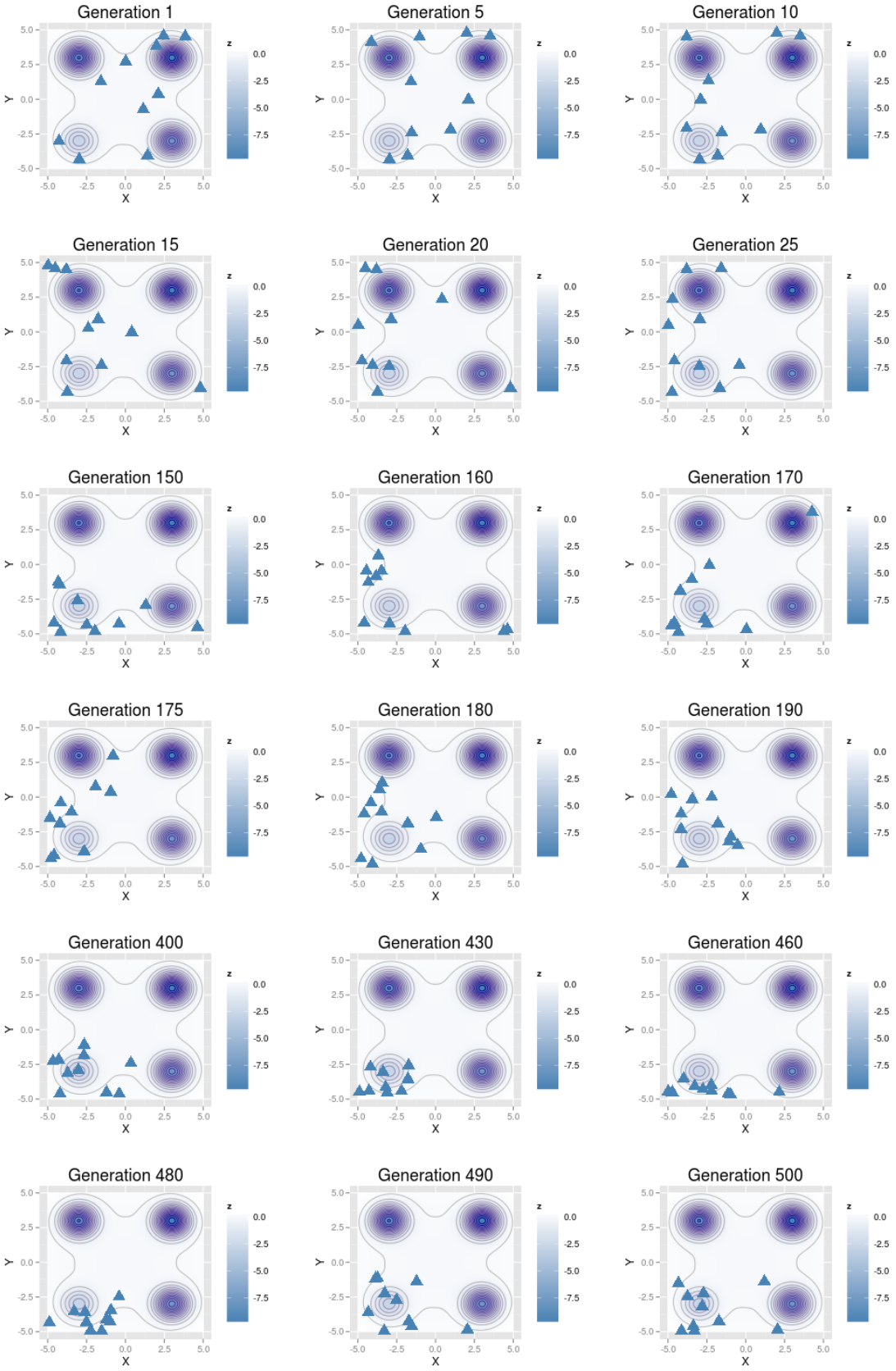


*Figure 3.1: Snap-shot-plots of $f_{MultMod}$ (Case 1). These plots show the position of the prey for selected generations in the evolution.*

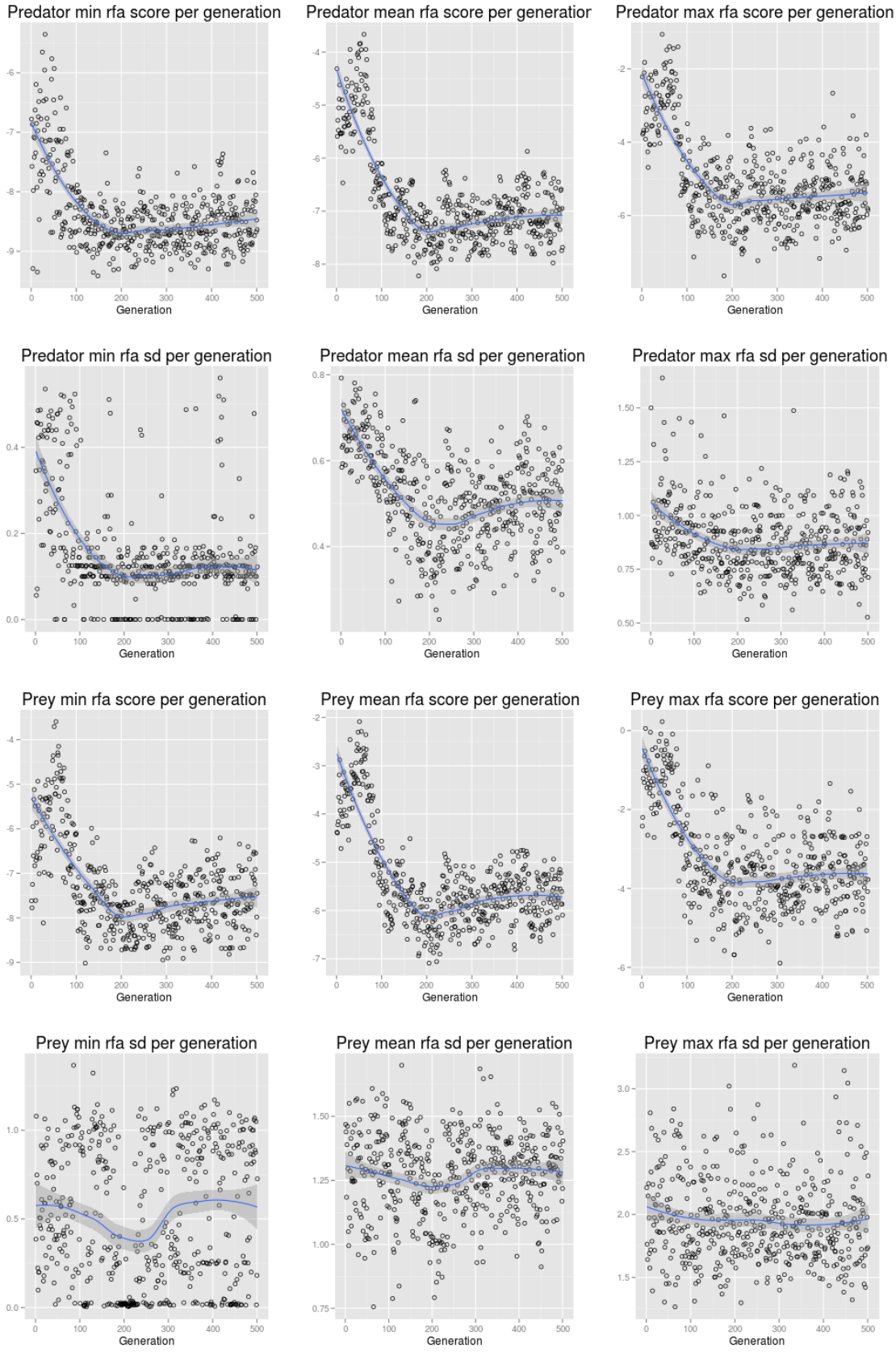


*Figure 3.2: History-plots (Case 1). Each of these plots shows a function applied to the rfa score or the rfa's score standard deviation on one species for each generation.*

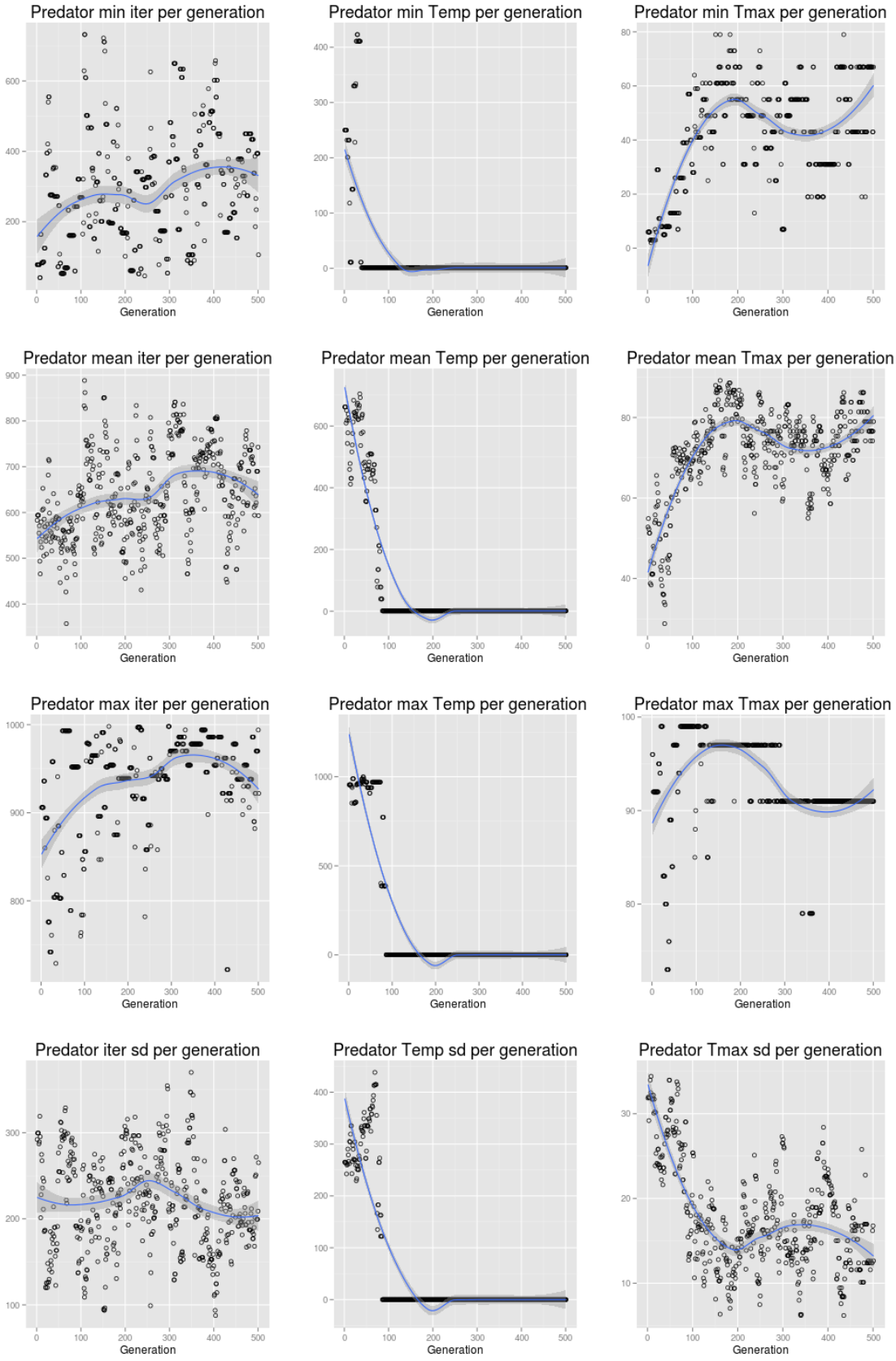


*Figure 3.3: Coevolution-plots (Case 1). Each of these plots shows a function applied to a gene from the individuals in the predator population for each generation.*

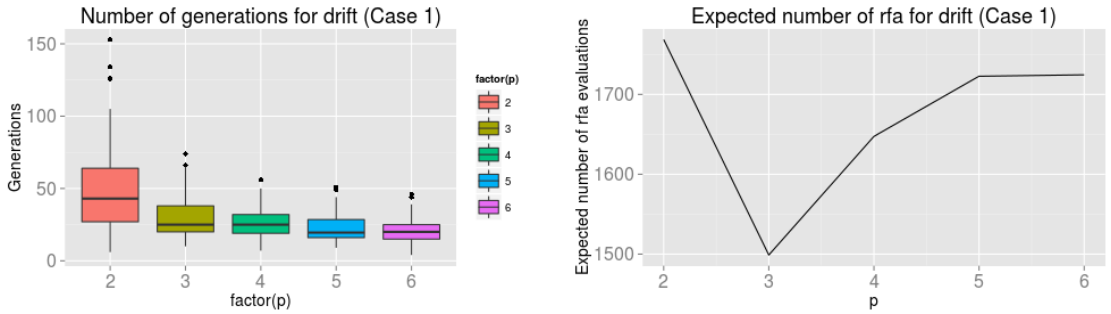


*Figure 3.4: Vary-p-boxplot and expected number of rfa (Case 1)*

### 3.1.2. Function $f_{\text{MultMod}}$ with SA & BFGS (Case 2)

The plots and results are again in accordance with the list presented in section 3.1. The following sections include plots and a short discussion on the results. The parameters tuned in SA are listed with bounds in table 3.1, the only addition is the max number of iterations for the BFGS algorithm. The allowed values are integers from 0 to 100.

#### Snap-shot-plots (Case 2)

We can see the results in figure 3.5. The setup is the same as for case 1, except BFGS has been added to follow SA. The figure is similar to figure 3.1, except that the prey is more concentrated in the prosperous region. The prey sometimes tries to escape but gets sucked right back.

#### History-plots (Case 2)

The results can be seen in figure 3.6. This is similar to figure 3.2, except we can clearly see the effect of the problems becoming harder in the beginning. With BFGS added, there is more consistency in the rfa scores.

#### Coevolution-plots (Case 2)

The resulting plot can be seen in figure 3.7. We can see the same behaviour in the temperature, it locks at the value 1. Here `tmax` gets locked at the value 90, since all the individuals gain that value, so it cannot budge. There seems to be some trend in the BFGS iterations.

#### Varying-$p$-plots (Case 2)

The results can be seen in figure 3.8.Here the results are even more drastic than in case 1. The expected number of generations it takes for all the prey to enter the prosperous region is significantly lower than for case 1 and the decrease in the mean for higher $p$ is not as significant. The expected number of rfa is a lower, with $p = 2$ giving the lowest expected number og rfa's.

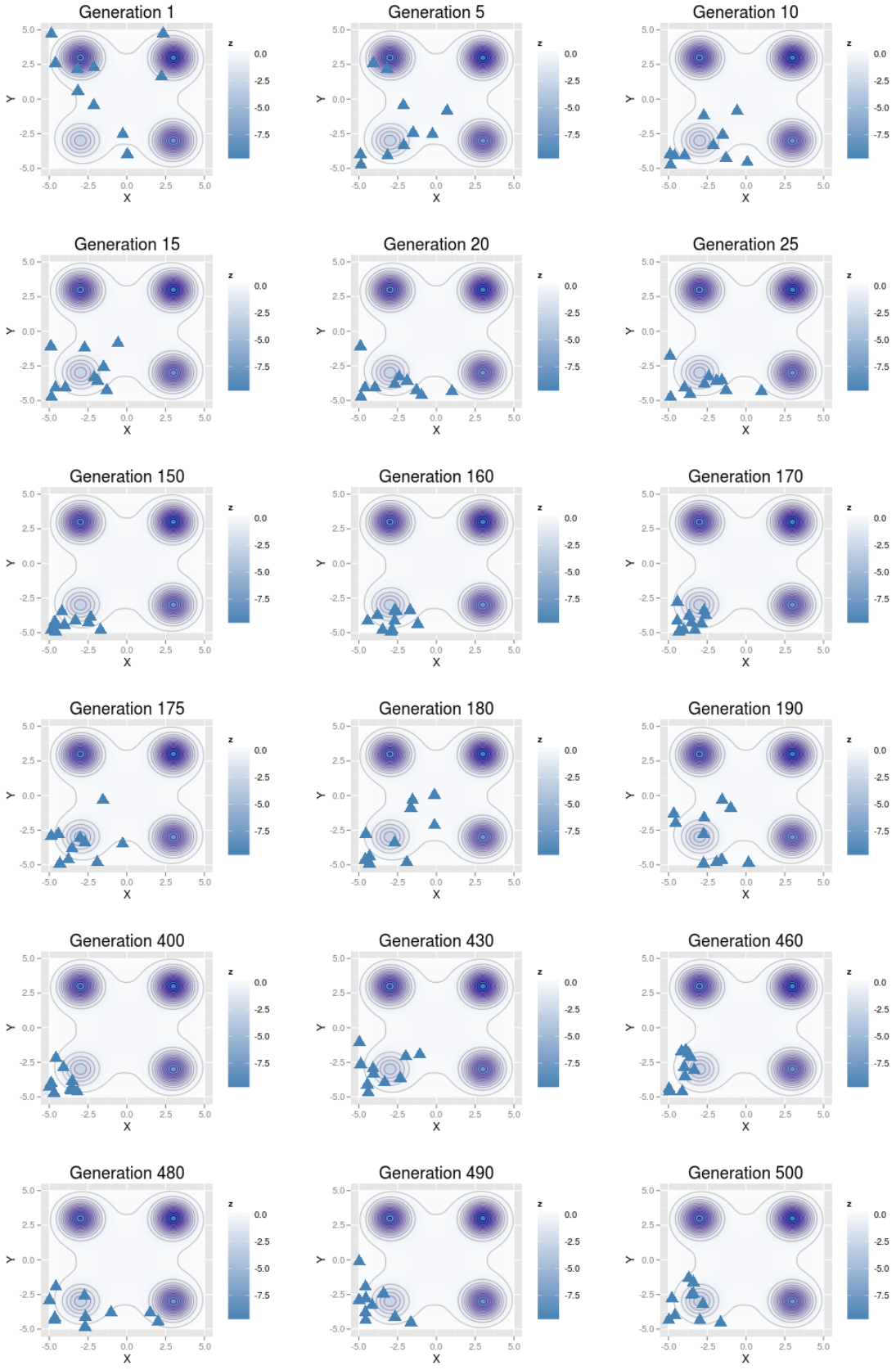


*Figure 3.5: Snap-shot-plots of $f_{MultMod}$ (Case 2). These plots show the position of the prey for selected generations in the evolution.*

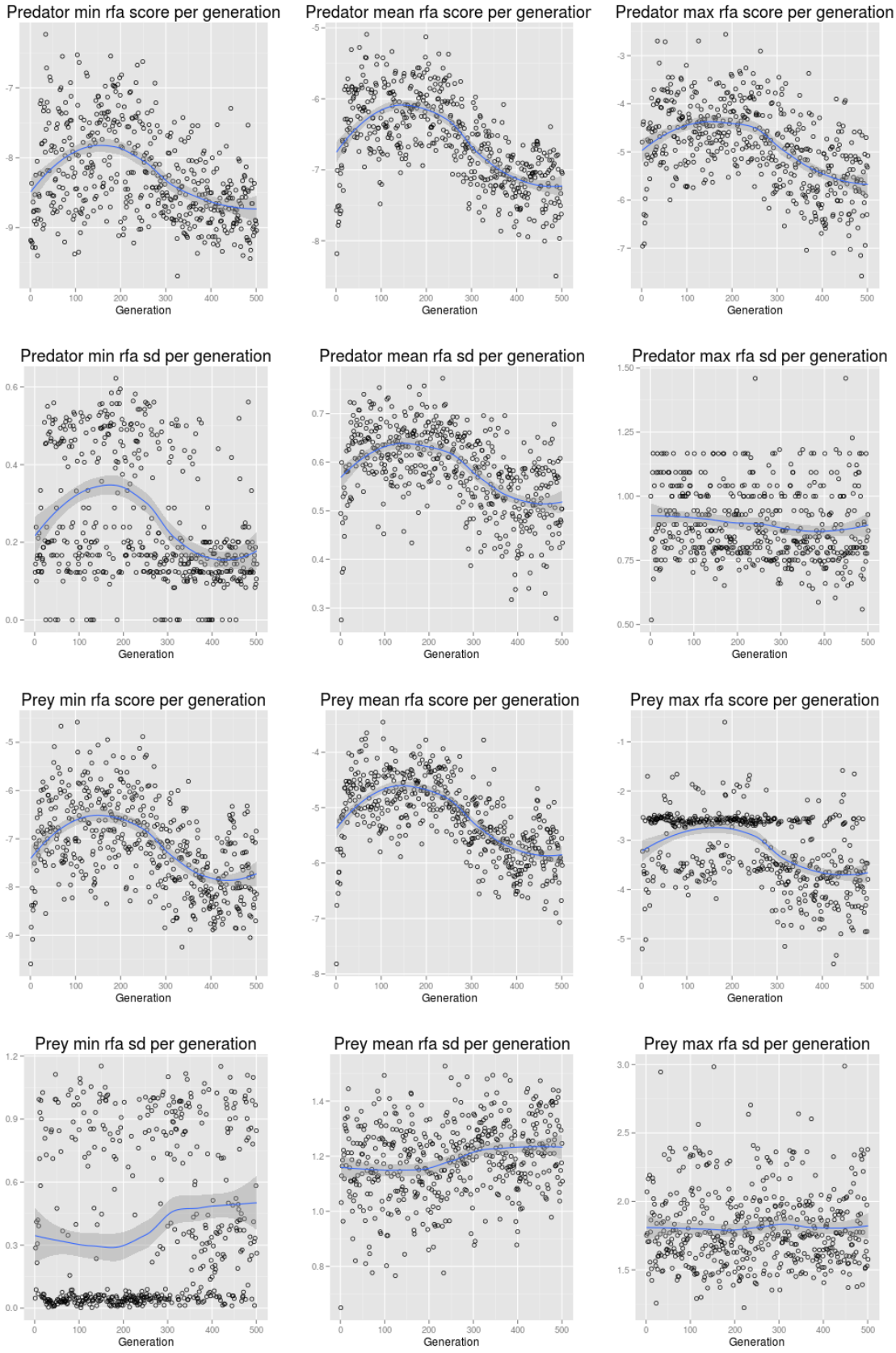


*Figure 3.6: History-plots (Case 2). Each of these plots shows a function applied to the rfa score or the rfa's score standard deviation on one species for each generation.*

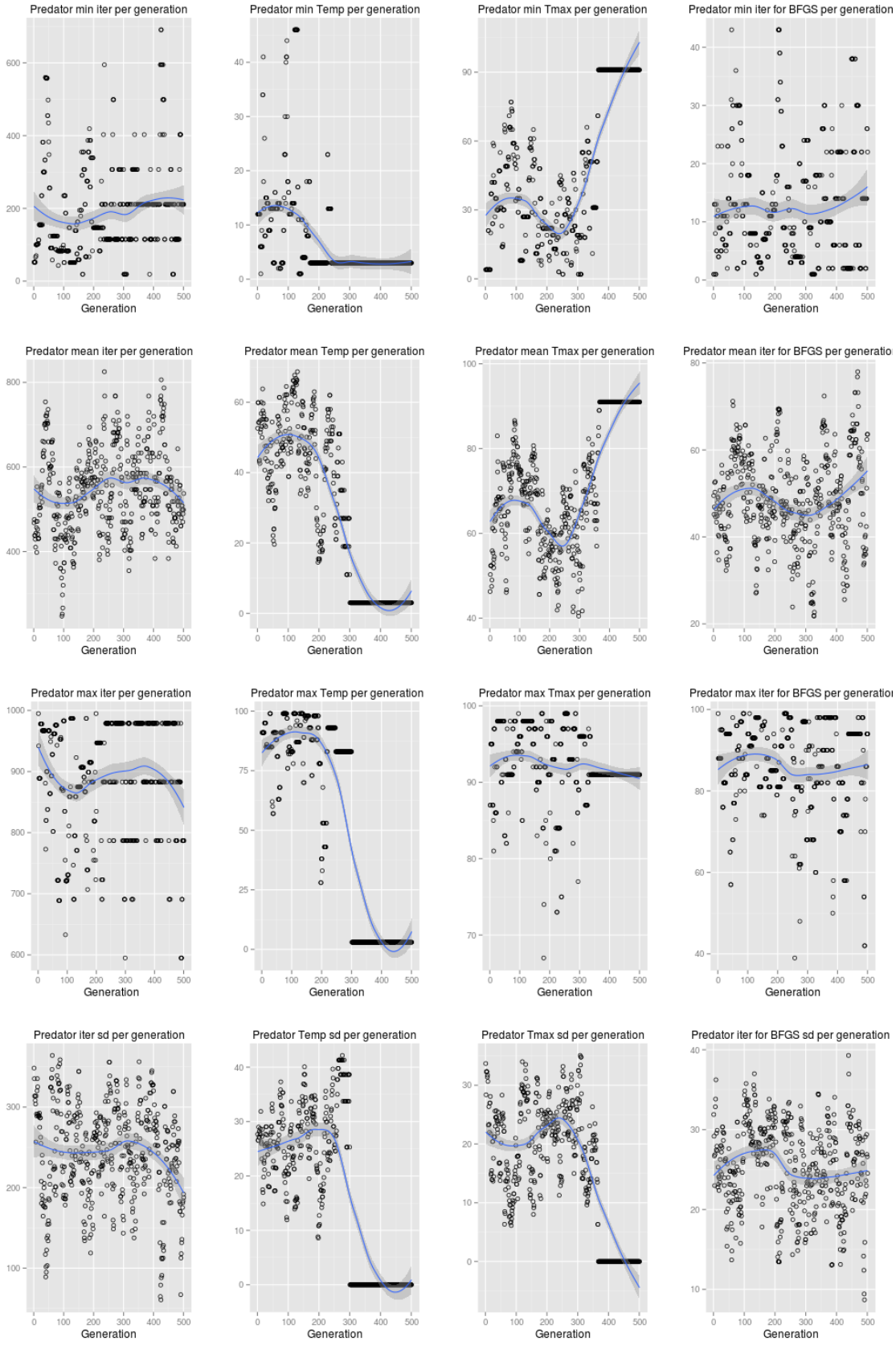


*Figure 3.7: Coevolution-plots (Case 2). Each of these plots shows a function applied to a gene from the individuals in the predator population for each generation.*

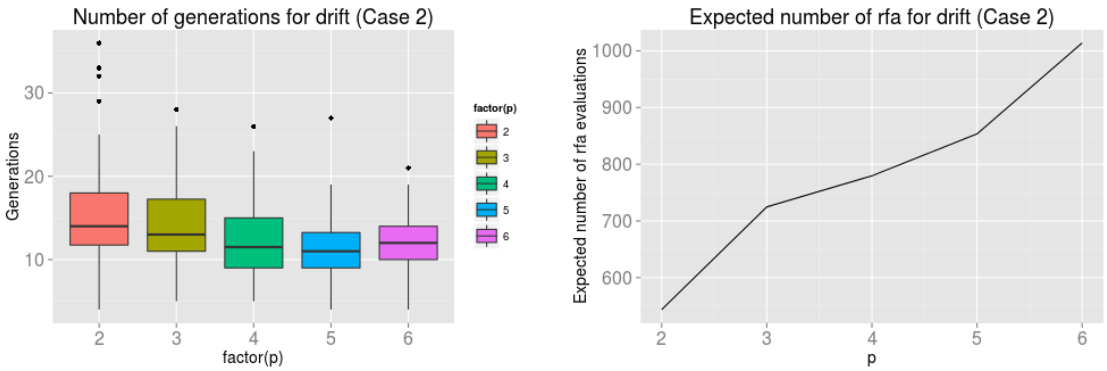


*Figure 3.8: Vary-p-boxplot and expected number of rfa (Case 2)*

### 3.1.3. Function $f_{\text{Ros}}$ with SA (Case 3)

The plots and results are in accordance with the list presented in section 3.1. The following sections include plots and a short discussion on the results. The parameters tuned in SA are listed with bounds in table 3.1, the only difference to case 1 is that the function is now $f_{\text{Ros}}$.

#### Snap-shot-plots (Case 3)

We can see the results in figure 3.9. The setup is the same as for case 1, except we have another function. The figure portrays similar behaviour to figures 3.1 and 3.5. There is a concentration in the prosperous region with some explorative outbursts from time to time.

#### History-plots (Case 3)

The results are presented in figure 3.10. We can clearly see the prey creating some very large outliers. There does not seem to be any obvious improvement in the Predator mean score (top middle), but this function is known to be notoriously hard for simulated annealing.

#### Coevolution-plots (Case 3)

The results are presented in figure 3.11. The results are similar to what is going on in figure 3.10. There are some fluctuations going on, the algorithm can't settle on anything specific, although it seems to favour more iterations. Again this function is hard for SA, so it is not surprising that we don't see the parameters for SA converge to some specific value.

#### Varying-$p$-plots (Case 3)

The results can be seen in figure 3.12. The effect here is quite different from $f_{\text{MultMod}}$. We have a lot more outliers and the drift and its variance increases with $p$, so naturally the expected number of rfa evaluations is lowest for $p = 2$.

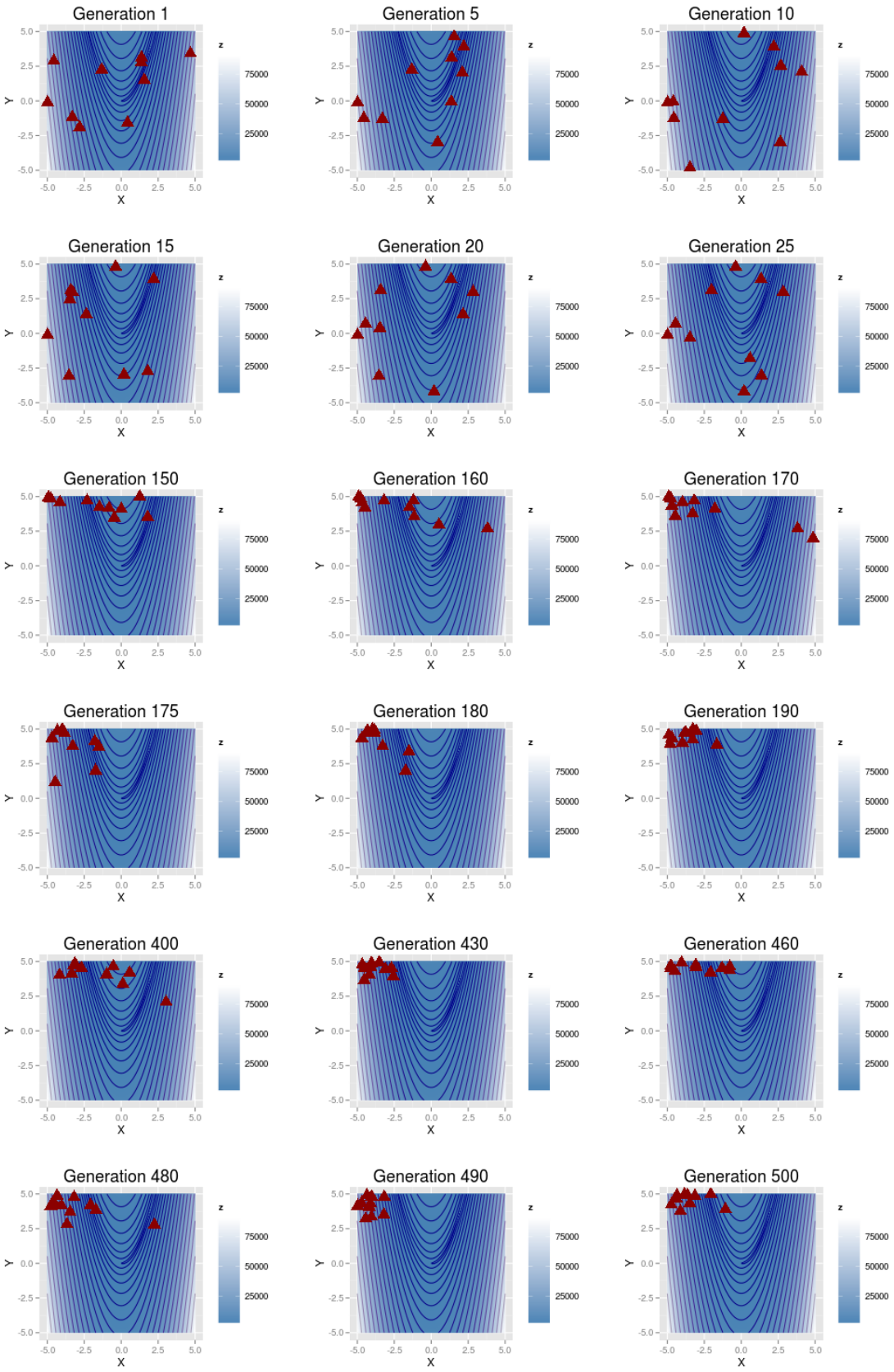


*Figure 3.9: Snap-shot-plots of $f_{Ros}$ (Case 3). These plots show the position of the prey for selected generations in the evolution.*

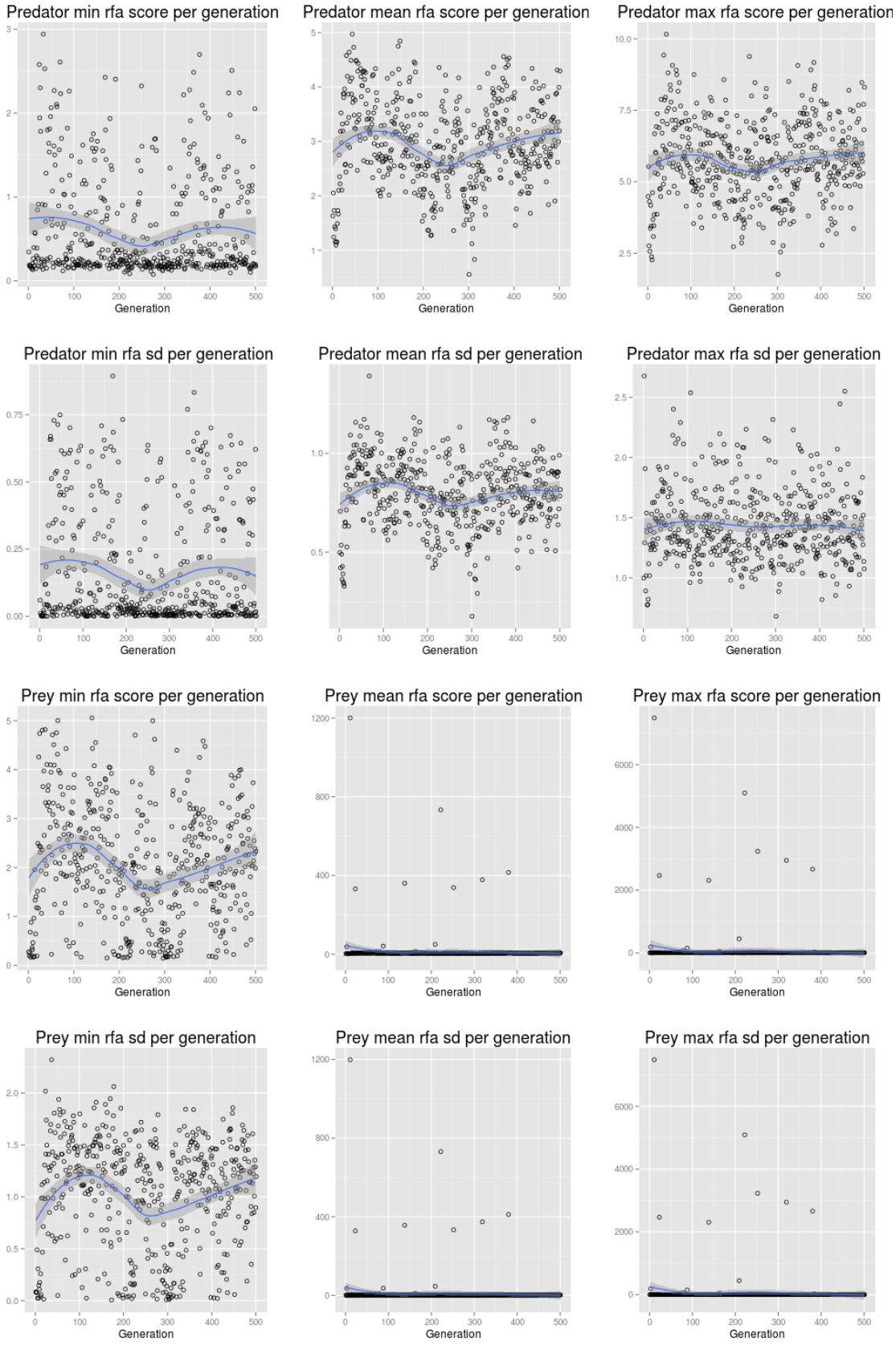


*Figure 3.10: History-plots (Case 3). Each of these plots shows a function applied to the rfa score or the rfa's score standard deviation on one species for each generation.*

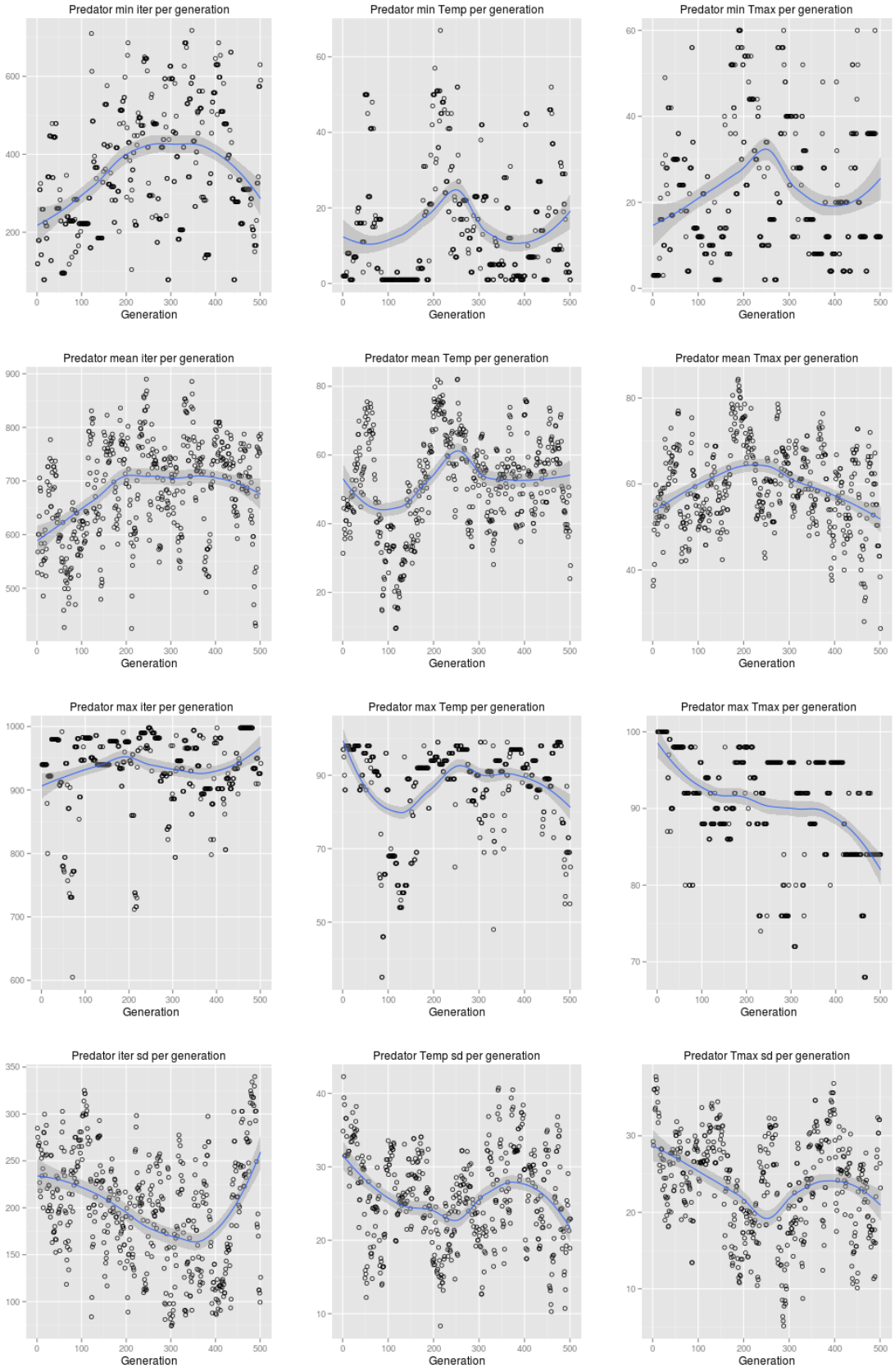


*Figure 3.11: Coevolution-plots (Case 3). Each of these plots shows a function applied to a gene from the individuals in the predator population for each generation.*

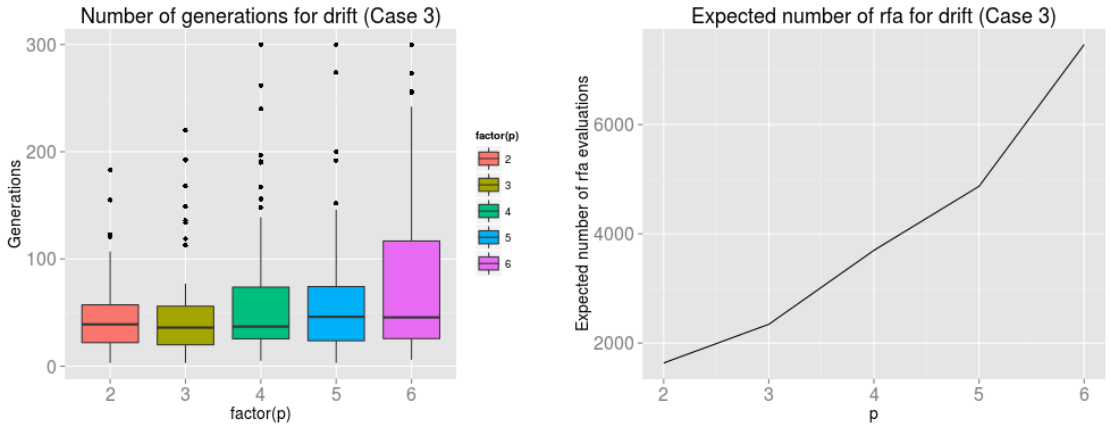


*Figure 3.12: Vary-p-boxplot and expected number of rfa (Case 3)*

### 3.1.4. Function $f_{\text{Ros}}$ with SA & BFGS (Case 4)

The plots and results are in accordance with the list presented in section 3.1. These following sections include plots and a short discussion on the results. The parameters tuned in SA are listed with bounds in table 3.1, the only difference from case 1 is that the function is now $f_{\text{Ros}}$ and we have added BFGS, which tackles this function extremely well.

#### Snap-shot-plots (Case 4)

We can see the results in figure 3.13. As mentioned above, BFGS handles the function $f_{\text{Ros}}$ very well, so we expect to see more fluctuations in the movement of the prey.

#### History-plots (Case 4)

We can see the results in figure 3.14. The results seem to be rather flat, but there was some tuning needed in the beginning, which is revealed in figure 3.15. If the median is explored instead of the mean for the predator rfa score, then the result is very homogeneous. The small narrow peaks are caused by outliers, but the algorithm robustifies against them.

#### Coevolution-plots (Case 4)

The results can be seen in figure 3.15. We can clearly see that the algorithm prefers BFGS over SA. The SA iterations plummet to almost nothing right away, while the BFGS iterations increase and their standard deviation drops, so the BFGS max iterations are concentrated around the higher values. Here we clearly see that the coevolution algorithm almost throws out SA and favours the right algorithm to solve this optimization problem efficiently.

#### Varying-$p$-plots (Case 4)

The results are presented in figure 3.16. The variance is quite a lot higher compared to the other *vary-p*-plots. The prey no longer has a sense of what is the prosperous region and simply roams more around.

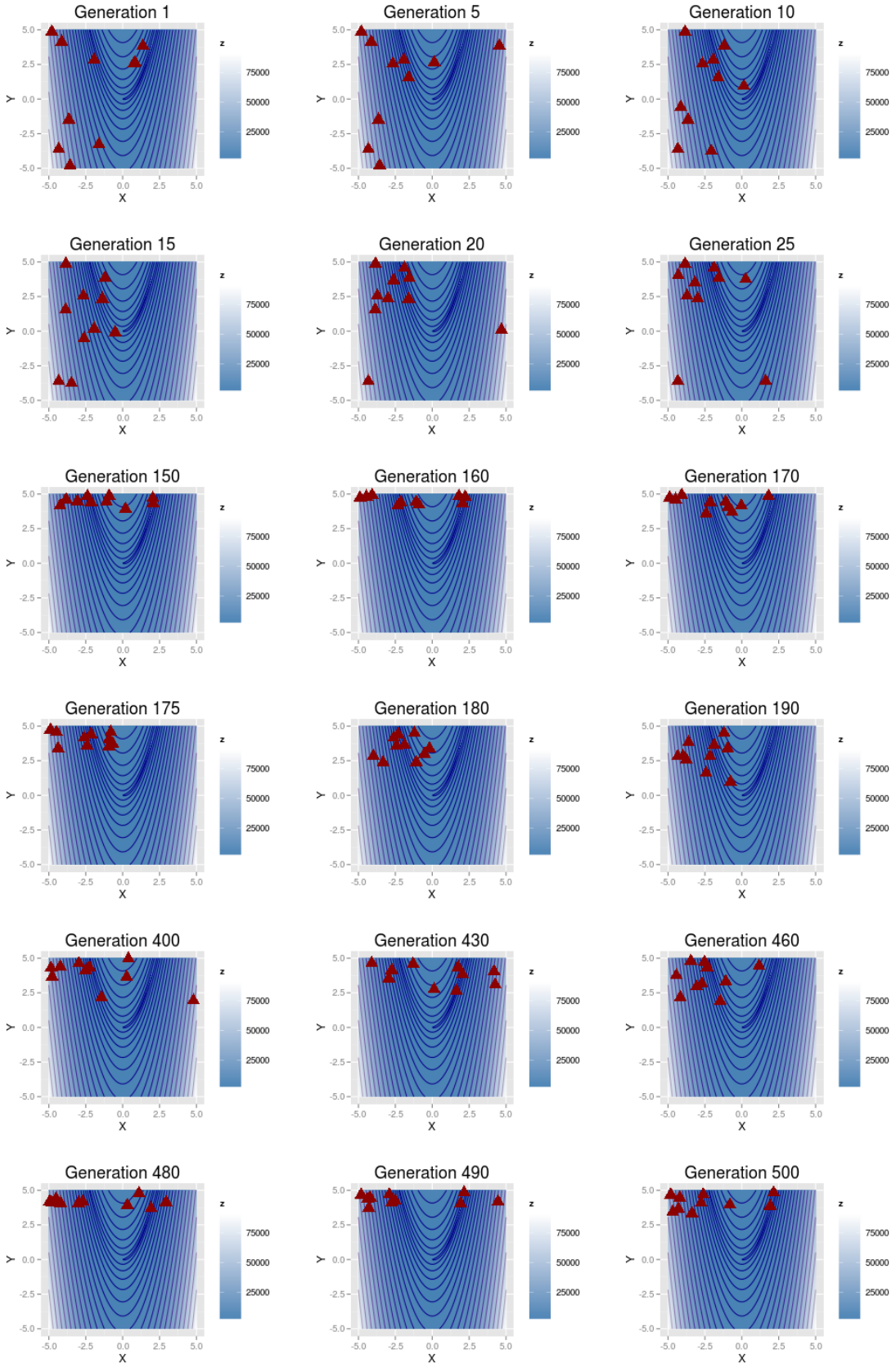


*Figure 3.13: Snap-shot-plots of $f_{Ros}$ (Case 4). These plots show the position of the prey for selected generations in the evolution.*

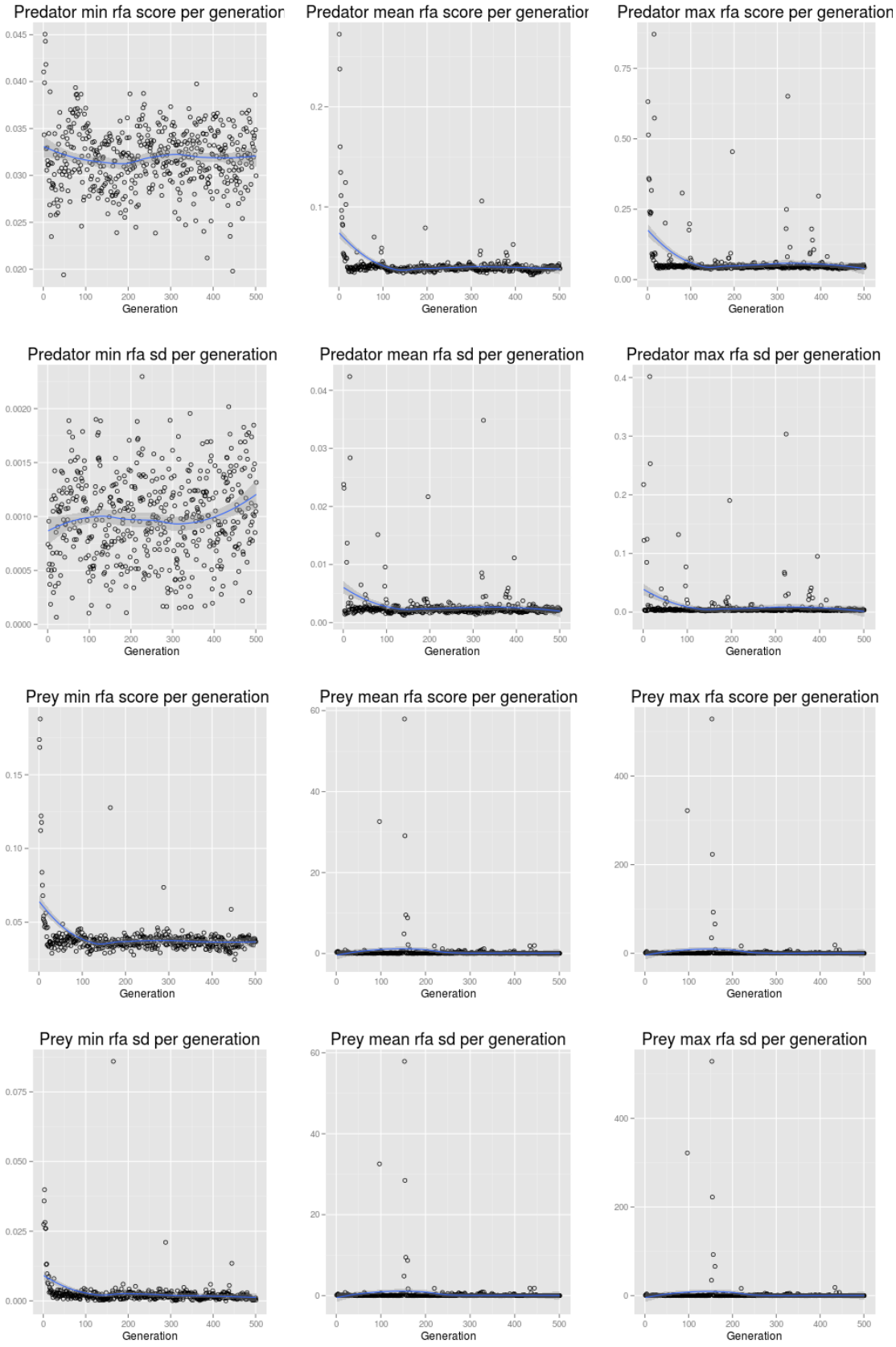


*Figure 3.14: History-plots (Case 4). Each of these plots shows a function applied to the rfa score or the rfa's score standard deviation on one species for each generation.*

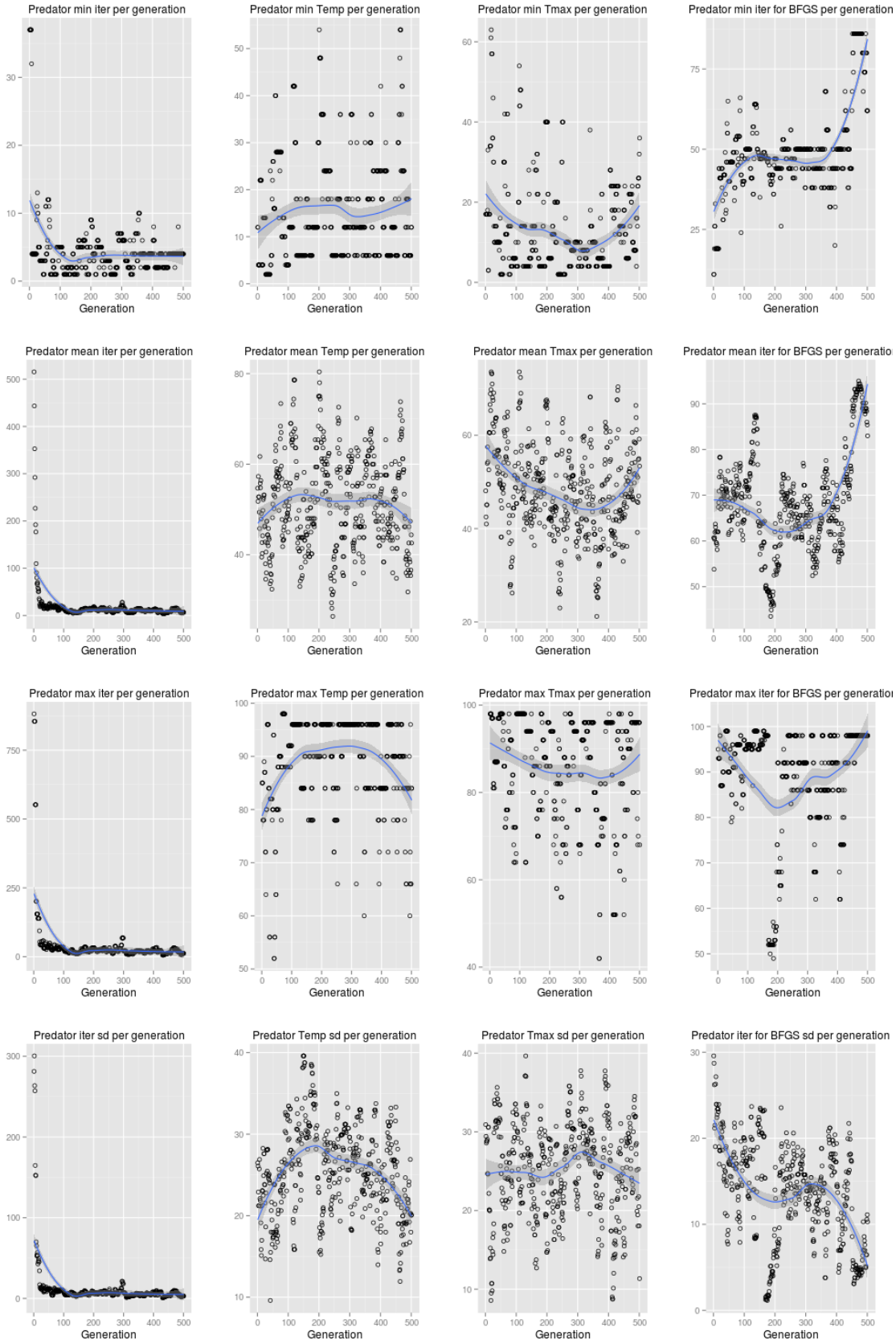


*Figure 3.15: Coevolution-plots (Case 4). Each of these plots shows a function applied to a gene from the individuals in the predator population for each generation.*

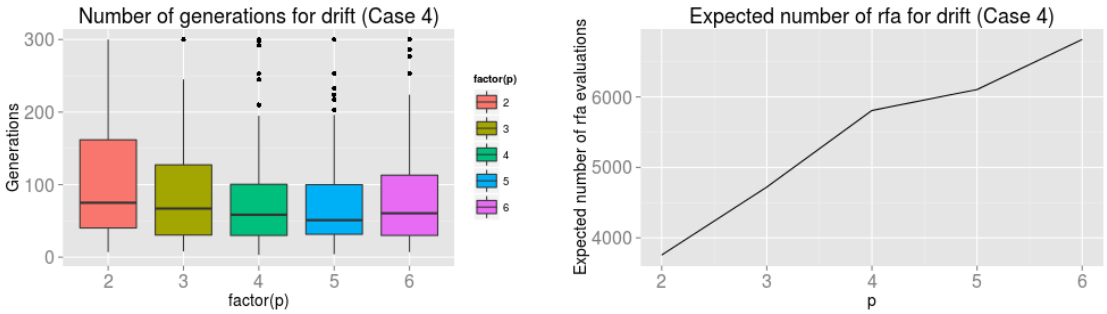


*Figure 3.16: Vary-p-boxplot and expected number of rfa (Case 4)*

### 3.1.5. Summary of Results for Auxiliary Problems

All the cases show learning to some extent. The only problem that does not show direct improvement for the predators is case 3, but that is no surprise, since the Rosenbrock function is notoriously hard for most heuristic methods and was especially designed for that purpose.

There is not so much difference between the first two cases, the most significant effect is that the mean rfa score for the predator plummets around the generation when the temperature reaches the value 1. This seems to be the most important contribution to the learning of the algorithm. It somewhat resembles a needle in a haystack search for the first problem but the learning gradient is not as steep in case 2. This is related to the fact that the step size is proportional to the temperature, so the SA algorithm naturally favours smaller step sizes for the function $f_{\text{MultMod}}$.

Case 4 is clearly distinct from case 2, (which is the other case that includes two algorithms as predators). In case 4 a clear distinction is made between SA and BFGS. BFGS is favoured, since it works extremely well on the Rosenbrock function, while the SA algorithm's iterations parameter plummets close to zero rather fast. The SA iterations do not get locked to some value as for some cases above, so it might be beneficial to have few iterations, so that BFGS does not start in the *worst* place. However all the places seem to become somewhat similarly hard when BFGS is applied.

The varying-$p$-plots and the expected number of rfa-evaluations-plots all tell the same tale. We favour lower values of $p$ for less rfa-function evaluations, still $p = 2$ shows significantly higher variance in the drift, and in case 1 we conclude that $p = 3$ is the best parameter. We can also see from the plots that the variance in the drift increases with better performing algorithms. The reason for this is mainly because when all the predators can handle all the prey, the prey no longer can differentiate as well between good and bad regions, its rfa-landscape becomes flatter. The main goal of robustifying algorithms is to flatten out the fitness-landscape of the parameters we are robustifying against.

#### As a diagnostics tool

We can see the results from applying BFGS to the prey from case 1 in figures 3.17 and 3.18. These results show that the BFGS algorithm is getting stuck at values significantly higher then what SA manages to achieve. Each point on figure 3.17 is the mean of the result from applying BFGS to the ten best prey individuals from each generation. If we look at figure 3.2, we can see that the score goes significantly

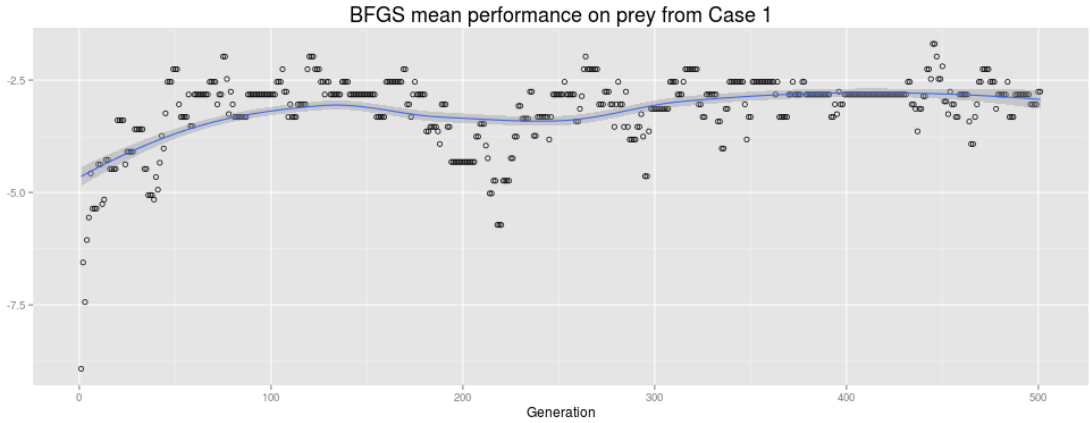


*Figure 3.17: Diagnostics-plot from Case 1 (Mean)*

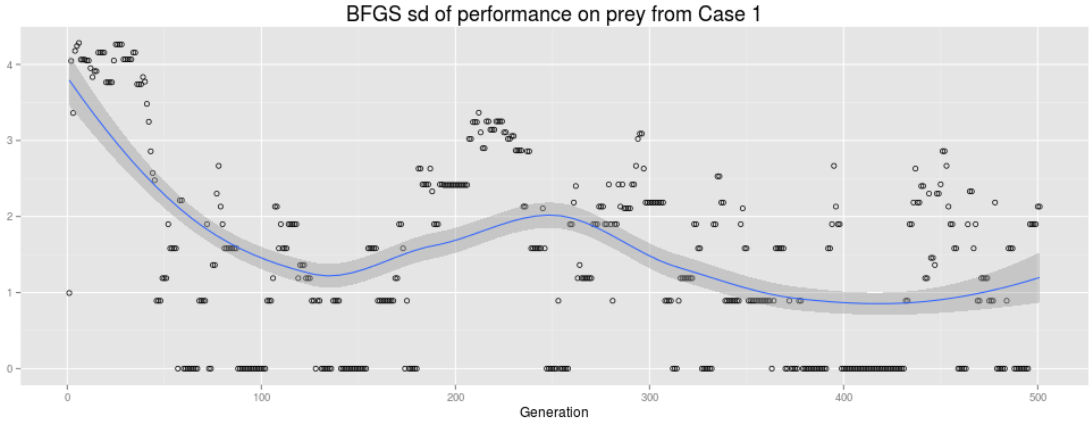


*Figure 3.18: Diagnostics-plot from Case 1 (Standard-deviation)*

lower. So this hints the existence of at least one other local optimum. The Hessian at that point reveals that it is indeed a minimum.

Here we knew of the other optimum points, but when we have little information on the function, this can help.

## 3.2. Results from gadget-runs

This section describes test runs using `gadget`. The model used in this case is the *haddock-example*. Further information about this model can be found on the MRI webpage (see appendix A), but explaining it is outside the scope of this thesis. The data are simulated, so these should be ideal conditions. The algorithm was tested with the bounds on the haddock parameters which are listed in table 3.2. These are 38 parameters, which corresponds to a modestly sized model.

*Table 3.2: Parameters for haddock-example*

| Parameter | Lower | Upper | Description |
|---|---|---|---|
| grq0 | 1 | 10 | $q_0$ in growth function |
| grq2 | 5 | 20 | $q_2, q_3$ in growth function |
| bbeta | 0.1 | 5000 | $\beta$ in beta-binomial |
| inage2 | 0.00001 | 1 | initial number of age 2 fish |
| inage3 | 0.00001 | 1 | initial number of age 3 fish |
| ⋮ | ⋮ | ⋮ | ⋮ |
| inage9 | 0.00001 | 1 | initial number of age 9 fish |
| rec78 | 0.2 | 34 | number of recruits in 1978 |
| rec79 | 0.2 | 34 | number of recruits in 1979 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| rec99 | 0.2 | 34 | number of recruits in 1999 |
| acomm | -100 | -1 | $\alpha$ in fleet suitability |
| bcomm | 0.1 | 10 | $\beta$ in fleet suitability |
| asur | -100 | -1 | $\alpha$ in survey suitability |
| bsur | 0.1 | 10 | $\beta$ in survey suitability |
| mult | 0.1 | 1 | multiplier for future fleet |

The parameters for the optimization methods are summarized in tables in appendix B, but the algorithms used for the predator chromosomes are SA, H & J and BFGS.

Some trial runs revealed problems concerning the application of the coevolution algorithm to this problem. The log-likelihood function is very ill-conditioned so the BFGS method is of no use unless it starts close to the optima. Therefore there is a threshold for which the other algorithms must pass before the BFGS algorithm becomes effective, they must lead BFGS to a region close to the optimum, where the likelihood function is almost quadratic. This is sort of a needle in a haystack problem and something needs to be done to counter against it, i.e. some of the

agents in the initial predator population must be able to find the optima. The log-likelihood function includes ridges and valleys similar to the ones of the Rosenbrock function, but the function is rather flat above it.

Another problem is the dimensionality of the species. We cannot just sample uniformly to create the initial predator individuals, unless if we are sure that at least some of them will be able to find the optima, but this makes the algorithm much more computationally intensive, since we would have to sample many individuals. This is what needs to be done to efficiently explore the effectiveness of the algorithm on larger problems but since we have some prior information, why not use it?

The adjustment made may seem somewhat *ad hoc*, but is needed to test the algorithm in a reasonable time and to be efficient. The predators are initially seeded from multivariate-uniform distributions where the support of the density only includes algorithms that can definitely tackle the problem, i.e. high values of iterators for small epsilon-values to ensure convergence. The bounds on the prey parameters are also tightened, although the lower and upper bounds from table 3.2. are used within `gadget`. The prey parameters are mainly tightened because, if the prey goes close to the bounds in `gadget`, they can be very troublesome, and we are not really interested in robustifying against starting values that are all close to or at the boundary.

One run of these initial predators can take around 1500 seconds on a standard desk computer (i7 quad core hyperthreaded). Therefore the number of agents was reduced to 8 and $p = 3$. The `inage` and `rec` parameters were all fixed at their optimum, to create a more streamlined test-bed that could be run in reasonable time. So we have 7 parameters that we are optimizing in `gadget`. The results plots can be seen in sections 3.2.1 and 3.2.2, the results are discussed in these sections. We shall explore plots similar to the history-plots and the coevolution plots from the auxiliary problem section. The weights in the weighted linear combination (see section 2.4.2) are chosen as 1 for $f_1$ (the final score from the optimization) and $\frac{1}{300000}$ for $f_2$ (the sum of iterations from the optimization algorithms). The value for the haddock model function at the optimum is $\approx 0.849$.

The parameters for the algorithms SA, H & J and BFGS were initially sampled uniformly within to the bounds which can be found in table 3.3. These parameters are further explained in appendix B. The upper and lower bounds on the number of iteration parameters are widened after the initial sampling of individuals.

These runs works mainly as a proof of concept, we wish see some trends in the evolution of the parameters in the optimization methods. Although there were 6 `gadget` processes running simultaneously in parallel the total time for this run was approximately 84 hours. The method is easily scalable, it can easily be run faster on a larger shared memory platform, if it is supposed to be run on a cluster then

some arrangement is needed on each node.

*Table 3.3: Parameter bounds on seeded predator for gadget-run*

| Parameter | Lower bound | Upper bound |
|---|---|---|
| simanniter | 30000 | 60000 |
| simanneps | 0.000001 | 0.1 |
| T | 190000 | 210000 |
| rt | 0.5 | 0.99 |
| nt | 2 | 100 |
| ns | 2 | 100 |
| vm | 1 | 10 |
| cstep | 0.3 | 0.9 |
| lratio | 0.1 | 0.5 |
| uratio | 0.7 | 0.9999 |
| hookeiter | 40000 | 60000 |
| hookeeps | 0.000001 | 0.00001 |
| rho | 0.61 | 1 |
| lambda | 0 | 0 |
| bfgsiter | 20000 | 25000 |
| bfgseps | 0.00001 | 0.01 |
| sigma | 0.01 | 0.95 |
| beta | 0.1 | 0.95 |
| step | 5 | 20 |
| gradstep | $10^{-6}$ | $10^{-2}$ |
| gradeps | $10^{-10}$ | $10^{-4}$ |

### 3.2.1. gadget-history-plots

The results are presented in figure 3.19. We can see the declining trend for the scores. The standard deviations is not changing much. Obviously a longer run is needed to get better parameter estimates. Because we started with parameters that can solve the problem, then the prey has a distorted sense of what is the prosperous region. Therefore one would wish to perform a run where the predator mean rfa score is obviously settling onto some constant. Still it is very promising to see that the learning gradient is non-constant, and there is obviously some enhancement going on. We can see a sudden change in the slope at around generation 8, that is when the number of iterations for simulated annealing stopped improving.

### 3.2.2. gadget-coevolution-plots

There are five coevolution-plots presented, showing the evolution of the optimization methods parameters.

Figure 3.20 shows the first four parameters from table 3.3. As expected the `simanniter` variable is decreasing but then becomes stable, while the other variables seem to scatter around uniformly.

Figure 3.21 shows parameters 5-8 from table 3.3. Some of the parameters do not change their value. The $\epsilon$ values for the methods were kept constant, only sampled in the beginning. Other parameters show some trends, such as the `vm` and `rt` parameters.

Figure 3.22 shows parameters 9-12 from table 3.3. The `lratio` and `uratio` parameters show some trends. `hookeiter` starts on its downward trend right away, it slows down when `simanniter` becomes stable.

Figure 3.23 shows parameters 13,15,16 and 17 from table 3.3. The `bfgsiter` clearly shows a trend towards fewer iterations and the `sigma` parameter seems to be settling on a value around 0.6.

Figure 3.24 shows parameters 18 and 19 from table 3.3. The `gradacc` and `gradeps` parameters were kept at their default values in `gadget`. The `beta` parameter and the `step` parameter show some trends.

A longer run is needed to get a fully robustified algorithm for `gadget`. That is when we see the iterations for the latter two algorithms become stable.

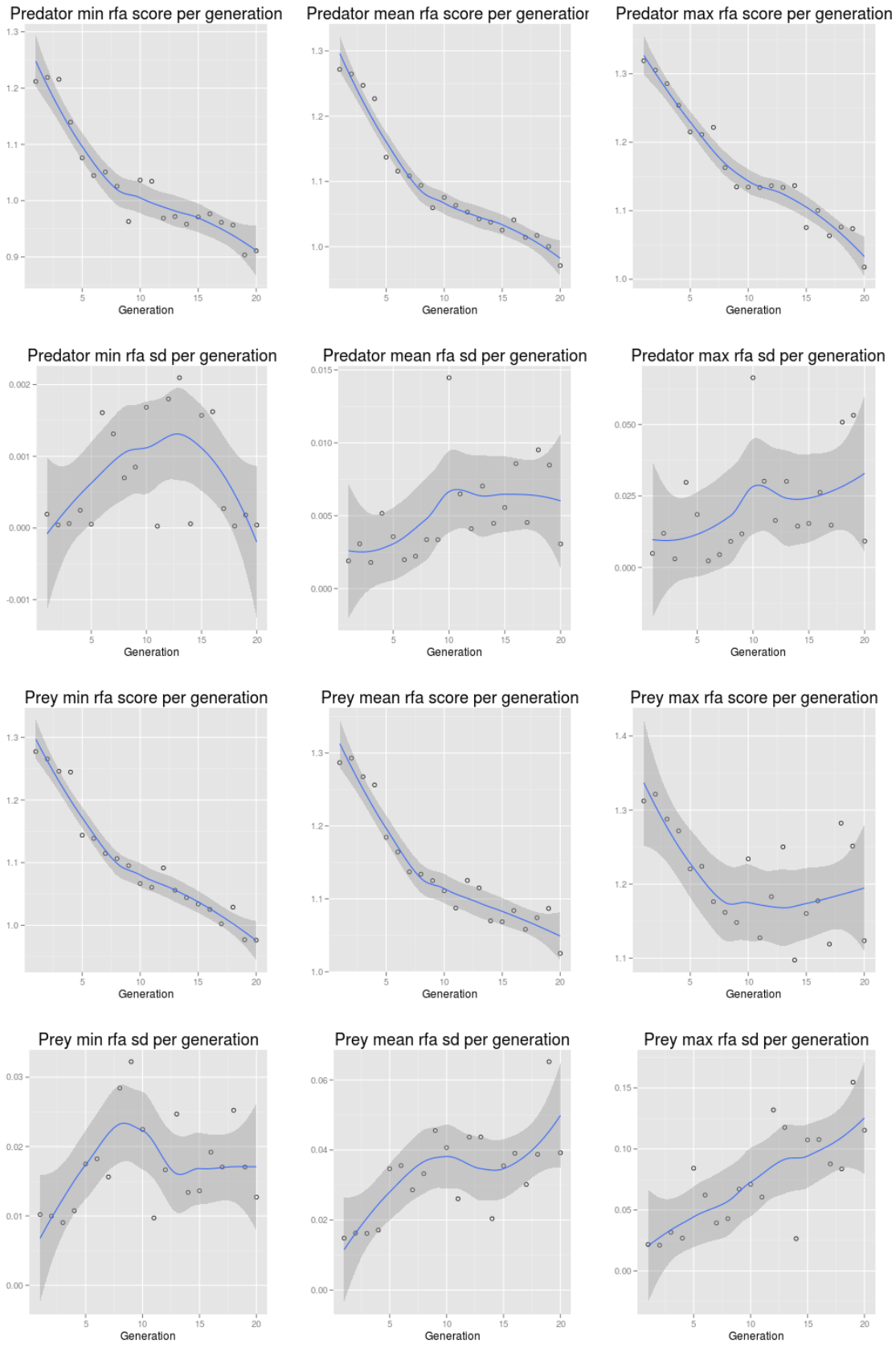


*Figure 3.19: History plot for* `gadget`

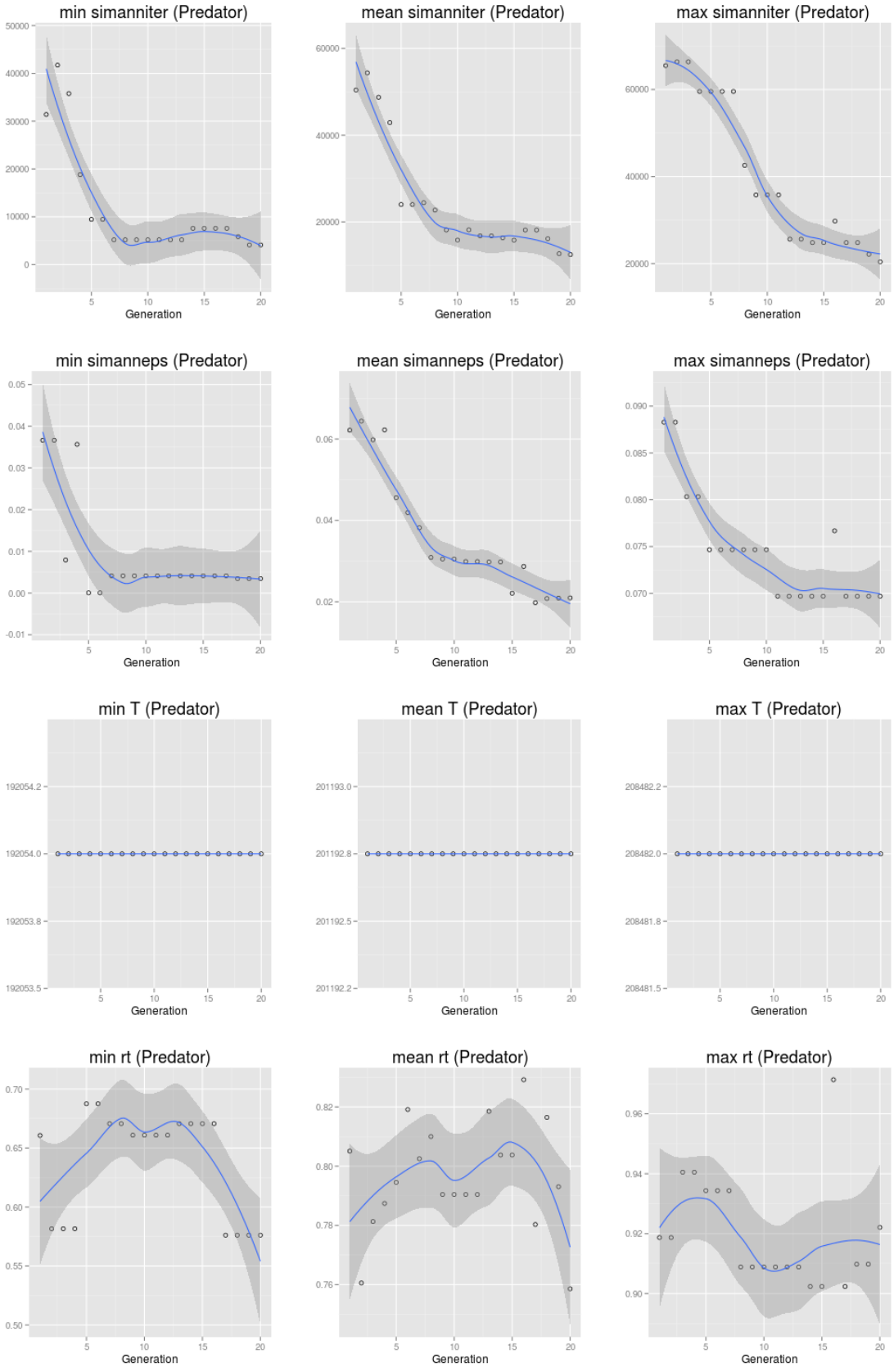


*Figure 3.20: First coevolution-plot for* `gadget`

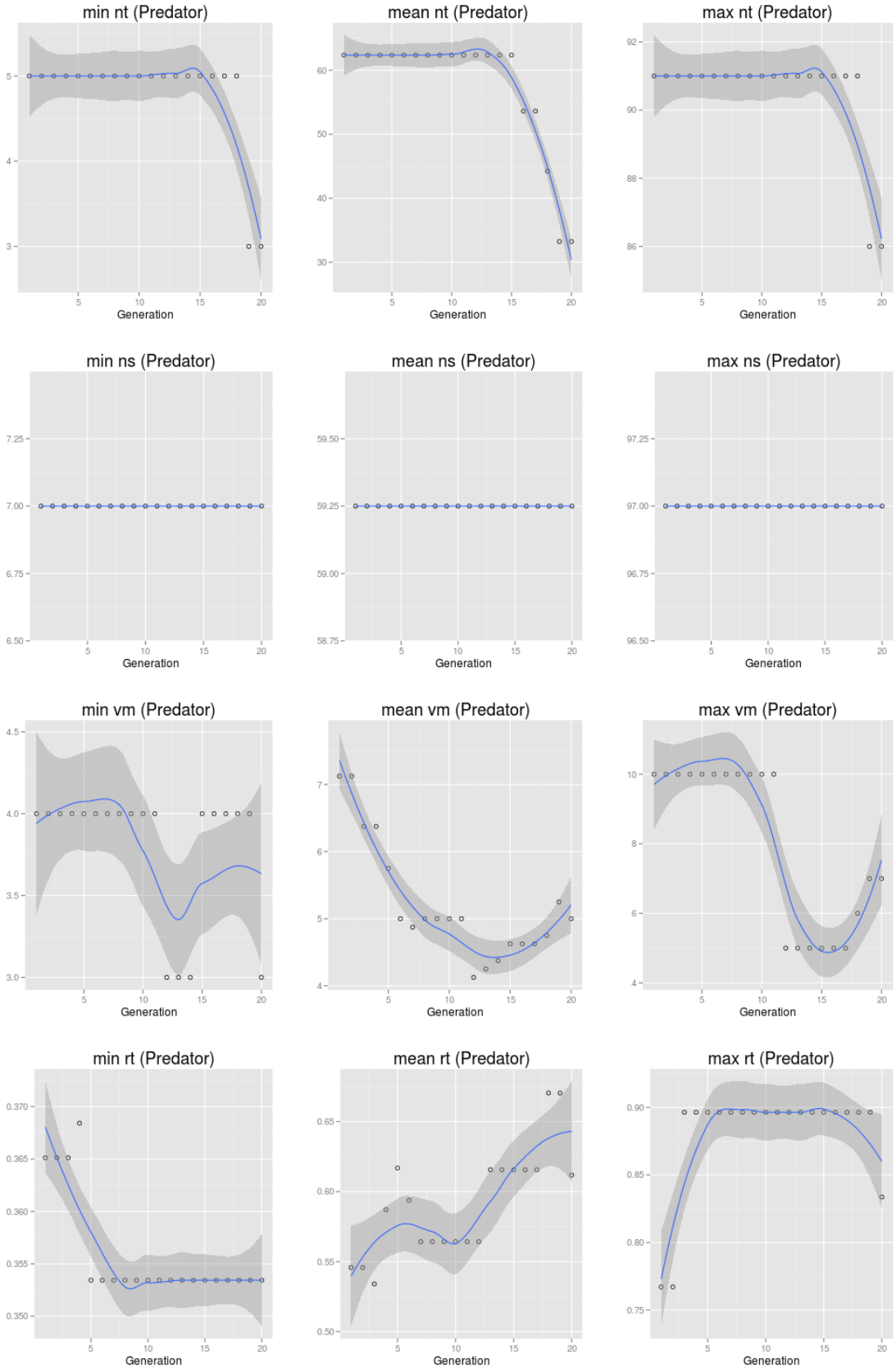


*Figure 3.21: Second coevolution-plot for* `gadget`

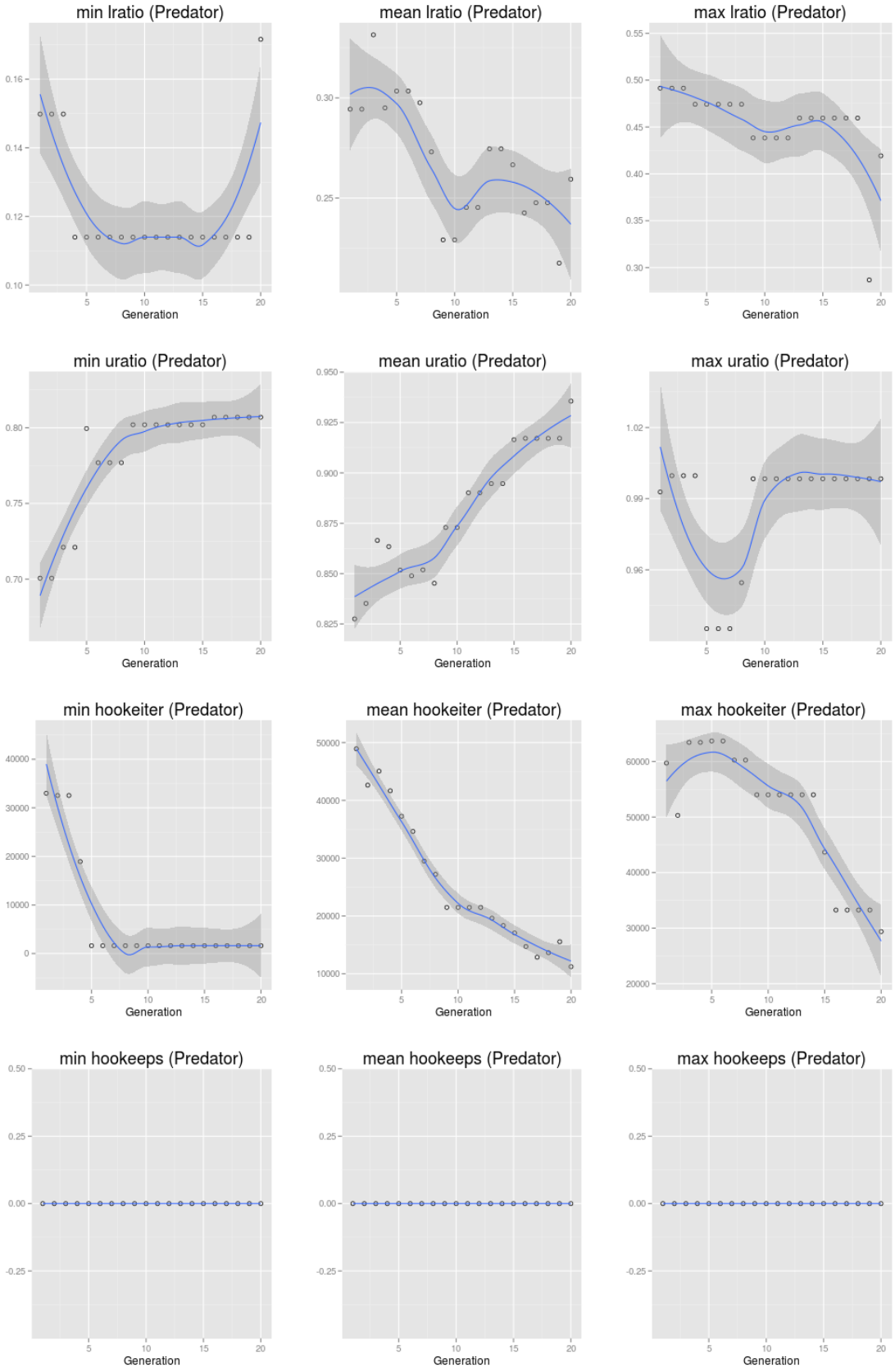


*Figure 3.22: Third coevolution-plot for* `gadget`

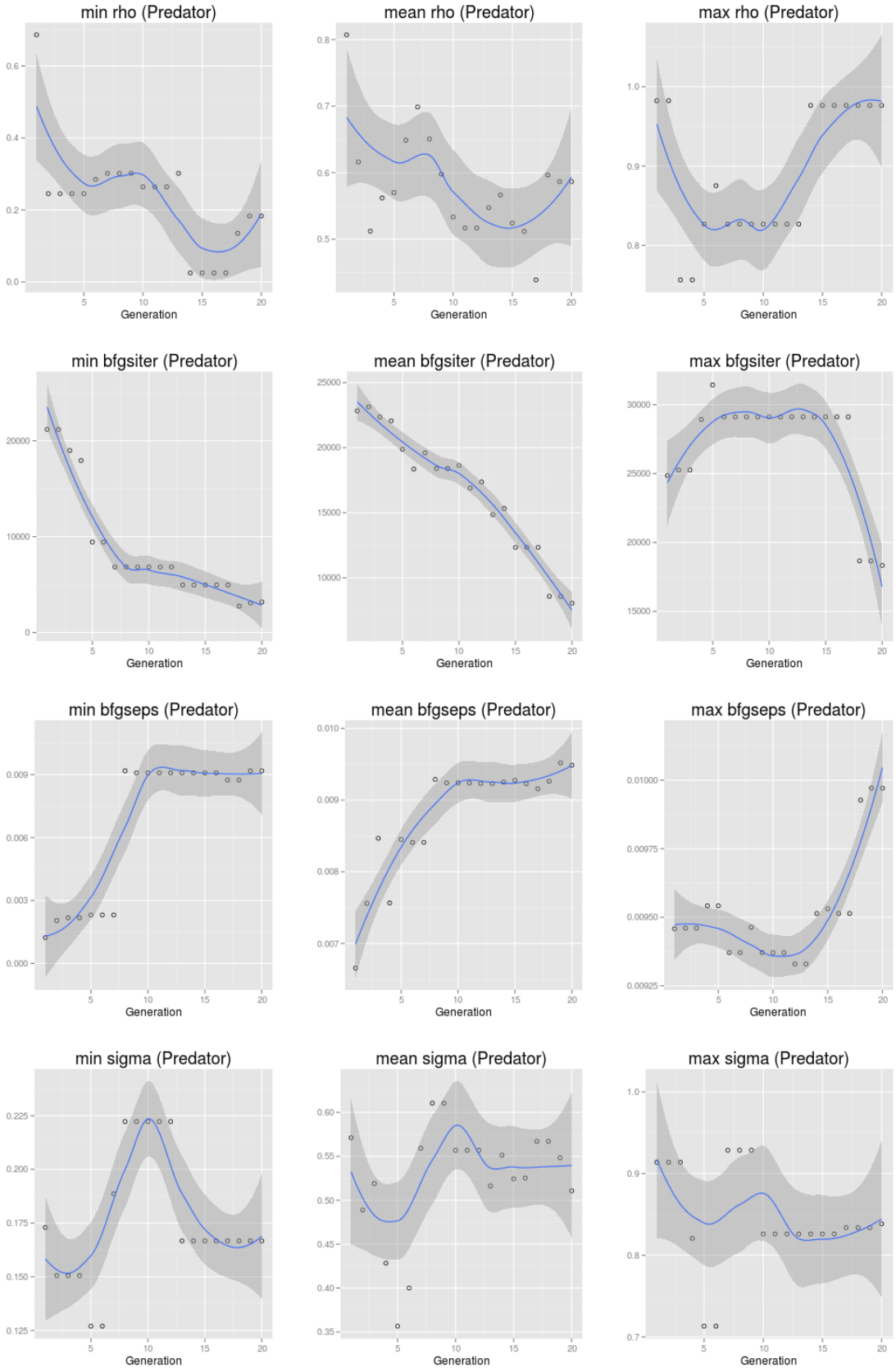


*Figure 3.23: Fourth coevolution-plot for* `gadget`

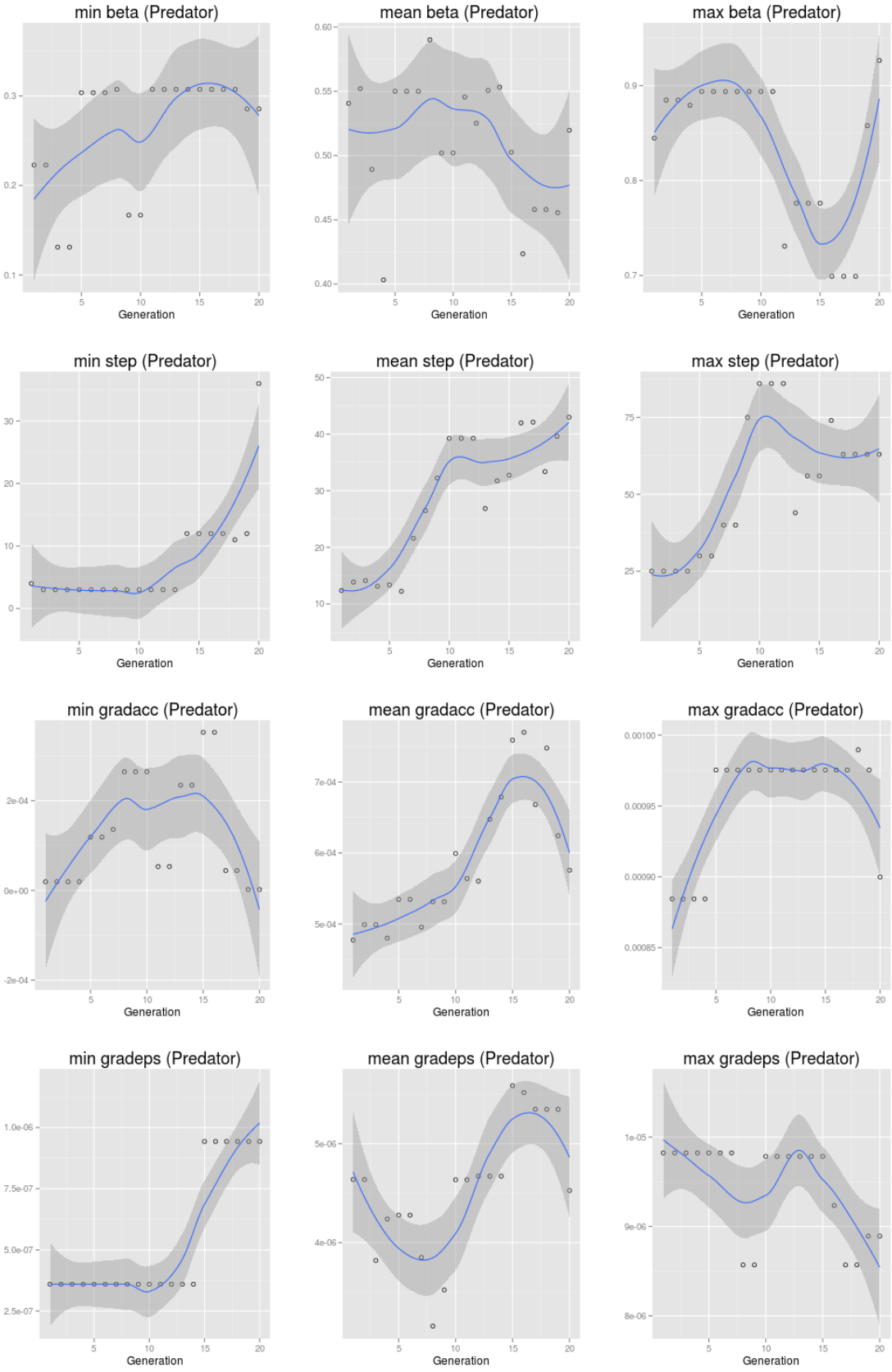


*Figure 3.24: Fifth coevolution-plot for* `gadget`

# 4. Discussion

The results from the auxiliary problems are quite convincing, but there are many aspects of the coevolutionary algorithm that were not explored in this thesis. More research is needed on many aspects of the algorithm.

More research is needed on efficiency with regards to the number of individuals, especially with respect to dimensionality. *A priori* information must be incorporated when the dimensionality increases too much and there must be something incorporated that gradually makes the prey more challenging, some kind of complexification as explained in Stanley et al. (2004).

Other ways of doing the relative fitness assessment must be explored as well. One idea is to use reinforcement learning (Sutton et al., 1998) instead of just an average. That could create some momentum for attraction to prosperous regions but needs to be explored further to realize its potential. Reinforcement learning is especially interesting with respect to drift.

More optimization algorithms should be explored as predators. The results from Case 4 (section 3.1.4) show the potential of the coevolutionary algorithm to deviate and choose the right algorithm for the underlying project. This could potentially be used to test claims from algorithm creators, whether their algorithms do indeed perform better on some test-beds.

More functions should be explored as prey. Many common functions can be found in the literature, which are used as test-beds to show performance of algorithms empirically. It would be interesting to see the algorithm choose strategies to solve NP-hard problems, such as 3-SAT.

Other variables should be explored for performing the relative fitness assessment. In this thesis the number of iteration and the final score from the optimization are used. One could also use distance from the true optima or other criteria that can create objectives.

The prey is of course not limited to starting positions in optimization. It can be a representative from a family of functions, e.g. something that parameterizes what function we are trying to optimize. This gives us extreme generality and one could

in theory use this to train strategies in games, if strategies can be represented as parameter vectors.

There are other ways of sampling new individuals that should be explored. Such as using an elitist strategy in the selection rather than just replacing parents by offspring. That is we choose the $N$ best individuals of the $2N$ present in the population when selection occurs. Other variants exist e.g. $(\mu + \lambda)$. A problem with this approach is that we do not incorporate the variance of the individuals in the relative fitness assessment, and this method may be more prone to following noise, yet one would have to deduce that empirically. One other method that could be incorporated is the variational number of individuals in each generation, e.g. if a parent produces a bad offspring that does not replace it for many generations, then it should be thrown out. Also if an individual is always being replaced by its offspring, it should be allowed to breed more than one offspring to get more individuals in that specific region.

Elitism could also be incorporated into the crossover/mutation phase, i.e. the generation of new individuals. A bias could be incorporated on individuals that have better rfa scores from last generations, such that they are more likely to be chosen for breeding than others. The rate of renewal for the species is not explored. Some control on this could enhance drift to a more narrow region, but it could also lead to the individuals getting stuck at bad values.

One obvious criticism on the choice of auxiliary problems is that the optimization methods are not exactly the same ones as those that are used in `gadget`. These methods serve exactly the same purpose and are very similar, so they should produce results similar to the methods in `gadget`. What is very much needed for `gadget`, though, is an implementation which is directly accessible from R, which allows testing of other optimization algorithms. DE looks e.g. very promising.

More theoretical results concerning convergence would be helpful, like the ones presented in Bélisle (1990), but the complexity of metaheuristics make it almost impossible to proof anything except in some simpler cases. What one would still aim for is to prove something concerning the collective behaviour of the populations. One would like to show that they move collectively and settle onto some region in the search space. Also one would like to show that the standard deviation in the population drops and that all the individuals represent the same phenotype(see table 2.1) or similar ones, of course this is not applicable to all cases.

In other instances where competitive coevolution has been used, the prey has only been used to create pathological hard cases, as described in Hillis (1990) and Luke (2009). Here the prey actually gives a byproduct that is of interest. We can potentially spot other local optima, that may not be expected in the problems we are solving. The method could also been used cooperatively in theory, where the

prey tries to find easier problems and helps the optimization methods settle upon an optimum.

The field of metaheuristics and evolutionary algorithms is growing fast. There are many new problems that people are working on and thanks to the fast evolution of modern processors, population based algorithms are becoming a more and more viable option, since they can easily be parallelized and scale well to large problems. The problems that arise in this thesis can generate many interesting subproblems to explore. One thing that needs to be done though is research on the collective behaviour of population-based algorithms and a way to quantify the effectiveness. The idea of introducing the notion of drift is a step in this direction. Another method of exploring this is looking at how the variation in the population changes with time. These methods are all very problem dependent, but showing results on some functions that portray important characteristics may give a hint of the general performance.

In the work presented in this thesis we have presented a method that will robust optimization algorithms for hard starting values. This method is implemented with two objectives, one of them is to minimize the number of iterations in the algorithms. This can in turn reduce the working time on certain models in scenarios where the optimization is performed repeatedly. This is exactly what is done in statistical bootstrapping (see e.g. Elvarsson et al. (2014)), which is a standard resampling method for estimating variance of model parameters.

# 5. Conclusion

A method for robustification of optimization algorithms against the problem of bad starting positions was developed. The difference from classic approaches lies mainly in the way the relative fitness assessment is performed. The main application of the method is to try to lower the time it takes to run the optimization procedures in the `gadget`-program (see appendix A).

The results from the coevolution algorithm are presented in chapter 3. First the algorithm was tested on two functions as the prey which portray interesting characteristics, namely multimodality and curved valleys; curved valleys are hard for most heuristics and non-gradient based methods. The predator optimizing algorithms that were made more robust were either only simulated annealing or simulated annealing and BFGS.

These smaller problems were run for 500 generations, the main objective was to see whether intransitivity would be an issue (De Jong, 2004). They portrayed what was to be expected. They managed to robustify the algorithms so that they performed better, and effectively chose the better algorithm for the job in cases where one outperformed the other (see section 3.1.4). There were no obvious signs of intransitivity, but when the algorithms achieve their optimal performance, then they have *won* and the prey loses its sense of fitness assessment, making it more explorative, but still it remains close to the old prosperous region.

The method can also be used as a diagnostics tool to detect multimodality. Some brief results on this are presented in section 3.1.5. One can see that the gradient based method repeatedly ends its optimization at values significantly higher than when a heuristic is combined with it which gives evidence of multimodality. One can estimate the Hessian at the point resulting from a BFGS run to be more sure of this.

The idea of drift is defined in section 2.5. It is a way to quantify the performance of the algorithm. The method is stochastic in nature, so the individuals don't converge to a single point, but a region. Drift is exactly what is needed to capture that performance. Of course one has to have extensive prior information about the problem at hand to be able to define a prosperous region for the given problem. Drift looks very promising for the problems in chapter 3.

Some preliminary results on `gadget` are presented in section 3.2. These were done as a proof of concept, to show that one can make the method work when dimensionality increases, while retaining few individuals in the populations. This was done by sampling individuals representing optimization algorithms that were capable of solving the problems. In this case the evolution showed gradual drop in the number of iterations.

The algorithm is programmed such that it performs the calculations in parallel. When the algorithm is run on `gadget`, the communication overhead becomes negligible, so the increase is almost linear for the number of cores used, although no figures portraying these results were shown.

This method is promising and has the potential to be used for other applications involving robustification of algorithms with respect to problems that can be defined by some parameter vector.

## 5.1. Future Work

Some of the problems and future work were presented in chapter 4. Some part of the future work is to do a longer run on a standard `gadget`-model to get good optimization parameter estimates. This is the most straight forward next step and the author plans to start these runs soon and hopefully get some results this summer.

Other things that need to be done is adding a population based search heuristic into `gadget`, instead of SA or in combination with it. The parallel version of SA is not good enough, and population based methods are easier to parallelize. Differential Evolution (Storn et al., 1995) looks promising, but other methods could potentially be added.

As mentioned in chapter 4, there are many variations on the coevolutionary algorithm that can be tweaked and explored. These could all lead to similar research as presented in this thesis. One easy generalization to explore is different number of predator and prey individuals per generation. More individuals in one species could potentially allow more of its individuals to reach a prosperous region faster. Another variant to this is a variable number of individuals; one could create a criterion where some parents create more offspring or more could die.

One thing that needs to be looked into is research on whether there is a relationship between the number of parameters in the problems and the number of individuals in the species with respect to drift. This could help choosing the initial number of individuals in the algorithm depending on the size of the problems.

# Bibliography


Stefansson, Gunnar. "Issues in multispecies models." Natural Resource Modeling 16, no. 4 (2003): 415-437.

Bradie, Brian. A Friendly Introduction to Numerical Analysis: With C and MATLAB Materials on Website. Person Prentice Hall, 2006.

Drton, Mathias, and Thomas S. Richardson. "Multimodality of the likelihood in the bivariate seemingly unrelated regressions model." Biometrika 91, no. 2 (2004): 383-392.

Begley, James, and Daniel Howell. "An overview of Gadget, the globally applicable area-disaggregated general ecosystem toolbox." ICES, 2004.

Holland, John H. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press, 1975.

De Jong, Edwin D. "Intransitivity in coevolution." In Parallel Problem Solving from Nature-PPSN VIII, pp. 843-851. Springer Berlin Heidelberg, 2004.

Samothrakis, Spyridon, Simon Lucas, Thomas Philip Runarsson, and David Robles. "Coevolving game-playing agents: Measuring performance and intransitivities." Evolutionary Computation, IEEE Transactions on 17, no. 2 (2013): 213-226.

Sörensen, Kenneth. "Metaheuristics—the metaphor exposed." International Transactions in Operational Research (2013).

Rosin, Christopher D., and Richard K. Belew. "New methods for competitive coevolution." Evolutionary Computation 5, no. 1 (1997): 1-29.

Storn, Rainer, and Kenneth Price. Differential evolution-a simple and efficient adaptive scheme for global optimization over continuous spaces. Berkeley: ICSI, 1995.

Stanley, Kenneth O., and Risto Miikkulainen. "Competitive coevolution through evolutionary complexification." J. Artif. Intell. Res.(JAIR) 21 (2004): 63-100.

Sutton, Richard S., and Andrew G. Barto. Introduction to reinforcement learning. MIT Press, 1998.

Luke, Sean. Essentials of metaheuristics. available at http://cs.gmu.edu/∼sean/book/metaheuristics/ 2009: p:109-114.

Hillis, W. Daniel. "Co-evolving parasites improve simulated evolution as an optimization procedure." Physica D: Nonlinear Phenomena 42, no. 1 (1990): 228-234.

Corana, Angelo, Michele Marchesi, Claudio Martini, and Sandro Ridella. "Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm Corrigenda for this article is available here." ACM Transactions on Mathematical Software (TOMS) 13, no. 3 (1987): 262-280.

Goffe, William L., Gary D. Ferrier, and John Rogers. "Global optimization of statistical functions with simulated annealing." Journal of Econometrics 60, no. 1 (1994): 65-99.

Hooke, Robert, and T. A Jeeves. "Direct Search Solution of Numerical and Statistical Problems." Journal of the ACM (JACM) 8, no. 2 (1961): 212-229.

Bertsekas, Dimitri P. "Nonlinear programming." (1999): 22-61.

Wolpert, David H., and William G. Macready. "No free lunch theorems for optimization." Evolutionary Computation, IEEE Transactions on 1, no. 1 (1997): 67-82.

Hansen, Nikolaus, Sibylle D. Müller, and Petros Koumoutsakos. "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)." Evolutionary Computation 11, no. 1 (2003): 1-18.

Goldberg, David Edward. Genetic algorithms in search, optimization, and machine learning. Vol. 412. Reading Menlo Park: Addison-wesley, 1989.

Glover, Fred. "Heuristics for integer programming using surrogate constraints." Decision Sciences 8, no. 1 (1977): 156-166.

Glover, Fred. "Future paths for integer programming and links to artificial intelligence." Computers & Operations Research 13, no. 5 (1986): 533-549.

Pólya, George. "How to solve it. Princeton." New Jersey: Princeton University (1945).

Simon, Herbert A., and Allen Newell. "Heuristic problem solving: The next advance in operations research." Operations research 6, no. 1 (1958): 1-10.

Geem, Zong Woo, Joong Hoon Kim, and G. V. Loganathan. "A new heuristic optimization algorithm: harmony search." Simulation 76, no. 2 (2001): 60-68.

Mahdavi, M., Mohammad Fesanghary, and E. Damangir. "An improved harmony search algorithm for solving optimization problems." Applied mathematics and computation 188, no. 2 (2007): 1567-1579.

Warnes, J. J., and B. D. Ripley. "Problems with likelihood estimation of covariance functions of spatial Gaussian processes." Biometrika 74, no. 3 (1987): 640-642.

Glover, Fred. "Tabu search-part I." ORSA Journal on computing 1, no. 3 (1989): 190-206.

Deb, Kalyanmoy. Multi-objective optimization using evolutionary algorithms. Vol. 2012. Chichester: John Wiley & Sons, 2001.

Bélisle, Claude JP. "Convergence theorems for a class of simulated annealing algorithms on Rd." Journal of Applied Probability (1992): 885-895.

Schoenauer, Marc, Ilya Loshchilov, and Michele Sebag. "Adaptive coordinate descent." In Genetic and Evolutionary Computation Conference (GECCO 2011), no. EPFL-CONF-186023, pp. 885-892. 2011.

Rosenbrock, Howard H. "An automatic method for finding the greatest or least value of a function." The Computer Journal 3, no. 3 (1960): 175-184.

Schmidberger, Markus, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, and Ulrich Mansmann. "State-of-the-art in Parallel Computing with R." Journal of Statistical Software 47, no. 1 (2009).

Urbanek, Simon. "multicore: Parallel processing of R code on machines with multiple cores or CPUs." R package (v: 0.1-7), URL http://cran. r-project. org/package= multicore (2011).

Begley, J. "Gadget user manual." Marine Research Institute, Reykjavik (2004).

Metropolis, Nicholas, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. "Equation of state calculations by fast computing machines." The journal of chemical physics 21, no. 6 (2004): 1087-1092.

Dennis Jr, John E., and Robert B. Schnabel. Numerical methods for unconstrained optimization and nonlinear equations. Vol. 16. Siam, 1996.

Elvarsson, B. Þ., L. Taylor, V. M. Trenkel, V. Kupca, and G. Stefansson. "A bootstrap method for estimating bias and variance in statistical fisheries modelling frameworks using highly disparate datasets." African Journal of Marine Science 36, no. 1 (2014): 99-110.

# A. Gadget

`gadget` stands for:

> **G**lobally applicable **A**rea **D**isaggregated **G**eneral **E**cosystem **T**oolbox.

The user manual (Begley, 2004), which contains most vital information, can be found on the MRI website:

> `http://www.hafro.is/gadget/`

The following is a quick summary from the webpage:

Gadget is a flexible and powerful tool for creating ecosystem models. The program was developed for modelling marine ecosystems in a fisheries management and biology context, however there is nothing in the program that restricts it to fish, and models have been developed to examine marine mammal populations. Indeed there is nothing to demand that the populations being considered are marine, or even aquatic, in nature.

Gadget allows you to include a number of features into your model: One or more species, each of which may be split into multiple stocks; multiple areas with migration between areas; predation between and within species; maturation; reproduction and recruitment; multiple commercial and survey fleets taking catches from the populations.

Gadget does two separate, but related things. Firstly it takes a model specification and performs a simulation using that set up. The model specification dictates the form of the equations to be used to describe growth, recruitment, fleet selectivity and so on, and the exact parameters to be used in these equations. Gadget will calculate model population and catches through time for your given set up. Note that to do this it does not use real-world data (except possibly overall catch tonnage). The program then compares various aspects of the modelled catches with real-world data from actual catches, and produces numeric likelihood scores measuring how well the model matched each data set. The program also computes a single overall likelihood score. This is a single number representing the 'goodness of fit' between the simulation and the data.

It is worth repeating this point. Gadget runs a complete simulation without reference to any data. It then compares the modelled and real catches, and produces a score evaluating the fit between the two. Gadget is a computer model. It is not intelligent in any way, and the optimisers are merely attempting to minimize a single one-dimensional measure of fit between the model output and the data. It is possible that the results thus produced may not be at all sensible. In particular the likelihood functions are designed to examine small differences between two sets of numbers, and may behave strangely where the model and the data are vastly different.

`gadget` is a command-line tool written in C++. It is maintained by the MRI and all the source code is available on their webpage.

# B. Optimization Methods in Gadget

## B.1. Simulated Annealing

The implementation of simulated annealing in `gadget` is the one that can be found in Corona et al. (1987).

The simulated annealing algorithm is controlled by several parameters within `gadget`. The parameters are summarized in the following table.

*Table B.1: Parameters in simulated annealing*

| **Parameter** | **Purpose** |
|---|---|
| `simanniter` | Number of simulated annealing iterations |
| `simanneps` | Minimum epsilon, simann halt criteria |
| `T` | Simulated annealing initial temperature |
| `rt` | Temperature reduction factor |
| `nt` | Number of loops before temperature adjusted |
| `ns` | Number of loops before step length adjusted |
| `vm` | Initial value for the maximum step length |
| `cstep` | Step length adjustment factor |
| `lratio` | Lower limit for ratio when adjusting step length |
| `uratio` | Upper limit for ratio when adjusting step length |
| `check` | Number of temperature loops to check |

The algorithm starts by performing a cycle of random moves along the coordinate directions where a new point $\mathbf{x}'$ is accepted according to the Metropolis criterion (Metropolis et al., 2004). The criterion is the following:

> If $\Delta f \leq 0$, then accept the new point $\mathbf{x}_{i+1} = \mathbf{x}'$
> else accept the new point with probability $p(\Delta f) := \exp(-\Delta f/T)$.

where $\Delta f = f(\mathbf{x}') - f(\mathbf{x}_i)$ and $T$ is the `t` parameter in the table called the temperature. The algorithm usually starts with high temperatures, then goes into cycles,

where the temperature is reduced accordingly. The algorithm can be described in the following steps as in Corona et al. (1987).

(1) Initialize the parameters.

(2) Perform a cycle of random moves, each along one coordinate direction. Accept or reject each point according to the Metropolis criterion. Record the optimum point reached so far. The new point is of the form:

$$\mathbf{x}' = \mathbf{x}_i + r v_{m_h} \mathbf{e}_h$$

where $v_{m_h}$ is the $h$-th coordinate of the step vector $\mathbf{v}$.

(3) Is number of cycles greater than or equal $N_s$? If yes continue to next step, else return to step 2.

(4) Adjust the step-length vector $\mathbf{v}$, it is adjusted such that approximately 50% of points are accepted, according to the formula

$$v_{m_i} = \begin{cases} v_{m_i} \cdot \left(1 + \frac{C(R_i - U)}{L}\right) & \text{if} \quad R_i > U \\ v_{m_i} \cdot \left(1 + \frac{C(L - R_i)}{L}\right)^{-1} & \text{if} \quad R_i < L \\ v_{m_i} \quad \text{otherwise} \end{cases}$$

where $R_i$ is the ratio of accepted moves for direction $i$. $C$ is the same as `cstep`,

(5) Is the number of step adjustments greater than or equal to $N_T$? If yes continue to next step, else return to step 2.

(6) Reduce the temperature, reset counter for number of step adjustments, set current point to the optimum.

(7) Is the stopping criterion satisfied? If yes continue to next step, else return to step 2.

(8) End.

The purpose of the algorithm is to perform a global search for the optimum, because the log-likelihood function may be ill-conditioned or multimodal, which can cause serious trouble for BFGS the quasi-Newton method. The log-likelihood functions created in `gadget` can include narrow valleys with flat steppes above, these steppes produce numerically unstable hessian matrices, so SA must manage to get down to these valleys.

## B.2. Hooke & Jeeves

The Hooke and Jeeves method is the one presented in Hooke et al (1961). It is what is called a *pattern search* and uses no gradient information, so it is not restricted to continuous optimization. The parameters controlling the method are summarized in table B.2.

*Table B.2: Parameters for Hooke and Jeeves optimization procedure*

| **Parameter** | **Purpose** |
|---|---|
| `hookeiter` | Number of Hooke and Jeeves iterations |
| `hookeeps` | minimum epsilon, Hooke and Jeeves halt criteria |
| `rho` | step length adjustment factor |
| `lambda` | initial value for the step length |

The method works can be described as follows. First the initial point is estimated, then a series of exploratory moves are tested in the coordinate directions with step length $\lambda$. If a better point is found, then it is set as the best point, else the old one is retained. If none of the coordinate directions yields better results, then we set $\lambda$ as $\rho\lambda$, i.e. we decrease the step size by a factor of $\rho$. In the end we quit because we cannot improve the solution by a halting criterion or the maximum number of iterations is reached.

## B.3. BFGS

The following is a short summary from the MRI webpage.

BFGS is a quasi-Newton optimisation method that uses information about the gradient of the function at the current point to calculate the best direction to look in to find a better point. Using this information, the BFGS algorithm can iteratively calculate a better approximation to the inverse Hessian matrix, which will lead to a better approximation of the minimum value.

From an initial starting point, the gradient of the function is calculated and then the algorithm uses this information to calculate the best direction to perform a linesearch for a point that is ”sufficiently better”. The linesearch that is used in Gadget to look for a better point in this direction is the ”Armijo” linesearch. The algorithm will then adjust the current estimate of the inverse Hessian matrix, and restart from this new point. If a better point cannot be found, then the inverse Hessian matrix is reset and the algorithm restarts from the last accepted point.

For a point at a stable minimum, the magnitude of the gradient vector will be zero, since there is no direction that the algorithm can move in to find a better local point. However, finding such a point using an iterative process can take an infinite number of steps, so the algorithm exits when the magnitude of the gradient vector is less than a small number. The current point is then accepted as being the 'solution'.

The "Armijo" linesearch calculates the stepsize that is to be used to move to a point that is "sufficiently better", along the search direction vector calculated from the gradient, to be $\beta^n$, where $n$ is the first integer that satisfies the Armijo rule given by the following inequality.

$$f(x) - f(x + \beta^n d) \geq -\sigma\beta^n d\nabla f(x)^T$$

where $d$ is the search direction vector and $\nabla f(x)$ is the gradient of the function at the current point.

In comparison to the other optimising algorithms that are currently implemented in Gadget, BFGS performs a local search, and so it doesn't cover the wide search area that the Simulated Annealing algorithm can use to look for an optimum. BFGS will usually take more computer time to find an optimum than the Hooke & Jeeves algorithm, since numerically calculating the gradient of the function is computationally very intensive. However, the optimum found by the BFGS algorithm will usually be better than that found by the Hooke & Jeeves algorithm, since a gradient search method is usually more accurate than a stepwise search method.

The BFGS algorithm used in Gadget is derived from that presented by Bertsekas (1999). The forward difference gradient algorithm used to calculate the gradient is derived from that presented by Dennis et al. (1996).

A summary of the parameter in the algorithm can be found in table B.3.

*Table B.3: Parameters for BFGS optimization procedure*

| **Parameter** | **Purpose** |
|---|---|
| `bfgsiter` | Number of bfgs iterations |
| `bfgseps` | minimum epsilon, bfgs halt criteria |
| `sigma` | *armijo* convergence criteria |
| `beta` | *armijo* adjustment factor |
| `gradacc` | initial value for gradient accuracy |
| `gradstep` | gradient accuracy adjustment factor |
| `gradeps` | minimum value for gradient accuracy |

# C. Parallelization of nested for-loops in R

There are many ways to parallelize independent `for`-loops in R, a rather recent summary of state of the ways to parallelize R code can be found in Schmidberger et al. (2009). The option used here was the `foreach` package which can use different parallel back-ends, in the experiments in this paper the `doMC` package was used as a parallel back-end and it serves as an interface for the `multicore` package which forks threads, the multicore package was published in 2011, so it is rather new, it can be found on CRAN (Urbanek, 2011).

The process of forking threads can create some overhead for the R process, if each call in the nested `for`-loop is quick relative to the overhead, then it might be more useful to just run the code sequentially. Here each call to `gadget` takes at least a couple of seconds, so the forking overhead is insignificant and we get almost ideal speed-up.

The following is a normal nested `for`-loop in R:

```
for(i in 1:N)
{
    for(j in 1:M)
    {
        Results[i,j]<-someFunction(Parameters(i,j))
    }
}
```

To parallelize this code we do the following:

```
library(foreach)
library(doMC)
library(multicore)
registerDoMC(numProc)
Res <- foreach(i = 1:N, .combine='rbind') %:% foreach(j = 1:N, .combine='c')%dopar%
{
    someFunction(Parameters(i,j))
}
Results <- Res
```

Note that inside the inner most loop, normally the output would be printed to the command-line, but the output is collected and combined via the `.combine` function defined in the header of the `for`-loop. Here `someFunction` is a wrapper for a `gadget` call, while `Parameters` are starting points for the optimization and optimization parameters.

It is possible to use another parallel back-end, such as `snow`, and run this on a cluster, but that would require some extra setup, such that all the file structures for a gadget run are available on all nodes, and `gadget` would have to be installed on all the nodes.

# D. Code for Coevolution methods

The following is the code to generate the competition matrix $C$ from section 2.4.2.

```
# This function creates the competition matrix
# which decides which agents from each group
# go against each other.
# N is the dimension of the square matrix
# p is a number which gives that each column
#   and each row will include between p and 2p
#   nonzero entries
genCompMat <- function(N,p)
{
  mat <- matrix(0,N,N)
  for(i in 1:N)
  {
    vecR <- sample(1:N,size=p)
    vecC <- sample(1:N,size=p)
    mat[vecC,i] <- 1
    mat[i,vecR] <- 1
  }
  return(mat)
}
```

The following is the code used to generate the inital individuals.

```
# This function creates the first generation of agents
# The output is a list of two matrices, the first being the
# Agents for the model and the second the agents for the
# optimization methods.
#
# numAg is the number of agents for each group.
# lower and upper params for f and Opt are bound vectors for
# the parameters
genAg <- function(N,lowerf,upperf, lowerOpt, upperOpt,F_vec)
{
  # The agents are column vectors, the last two values are
  # placeholders for average score and standard deviation.
  modelAg <- matrix(runif((length(lowerf)+2)*N,
                          min=c(lowerf,100000000,100000000),
                          max=c(upperf,100000000,100000000)),
                    length(lowerf)+2,N)
  # Same goes for the Opt params
  OptAg <- matrix(runif((length(lowerOpt)+2)*N,
                        min=c(lowerOpt,0,0),
                        max=c(upperOpt,0,0)),
                  length(lowerOpt)+2,N)
  for(i in 1:length(lowerOpt))
  {
    if(F_vec[i]==1)
    {
      OptAg[i,] <- ceiling(OptAg[i,])
    }
  }
  mat <- list(modelAg,OptAg)
  return(mat)
}
```

The following function is used to create a new population.

```
  # This function creates all generation of agents except the
# first one.
# The output is a list of two matrices, the first being the
# Agents for the model and the second the agents for the
# optimization methods.
#
# N is the number of agents for each group, N is not
# a parameter at the moment, but will maybe be in the future.
# lower and upper params for f and Opt are bound vectors for
# the parameters, we need this so we don't go out of bounds.
# F_par is the F in DE, F_vec is similar but a vector, where
#   the value is 1 for integer parameters.
NewAg <- function(Agents,lowerf,upperf, lowerOpt, upperOpt, CR, F_par, F_vec)
{
  modelAg <- Agents[[1]]
  optAg <- Agents[[2]]

  newModelAg <- compNewPop(modelAg, CR, F_par, lowerf, upperf)
  newOptAg <- compNewPop(optAg, CR, F_vec, lowerOpt, upperOpt)

  newAgents <- list(newModelAg, newOptAg)
  return(newAgents)
}
```

The main ideas in this function are explained in section 2.3.1.

```
# Function used in NewAg to generate the populations based
# on DE principles.
compNewPop <- function(agents, CR, F_par, lower, upper)
{
  numAg <- dim(agents)[2]
  newAgents <- agents
  for(i in 1:numAg)
  {
    x <- agents[,i]
    n <- length(x)
    xScore <- x[n-1]
    xSd <- x[n]
    vecABC <- sample((1:numAg)[-i], size=3)
    A <- agents[,vecABC[1]]
    B <- agents[,vecABC[2]]
    C <- agents[,vecABC[3]]
    R <- sample(1:(n-2),size=1)
    if(length(F_par)>1)
    {
      crossAg <- A
      # Incase we have integers, we need F_par to be one.
      for(j in 1:(n-2))
      {
        crossAg[j] <- crossAg[j]+F_par[j]*(B[j]-C[j])
      }
    }else
    {
      crossAg <- A+F_par*(B-C)
    }
    newAg <- x
    for(j in 1:(n-2))
    {
      r_par <- runif(1)
      if(r_par < CR | j==R)
      {
        if(crossAg[j] > lower[j] & crossAg[j]<upper[j])
        {
          newAg[j] <- crossAg[j]
        }
      }
    }
    newAg[n] <- 0
    newAg[n-1] <- 0
    newAgents[,i] <- newAg
  }
  return(newAgents)
}
```

This function performs the relative fitness assessment explained in section 2.4.2.

```
calcScore4x4 <- function(f,scoreFunc,bestAgents,Agents,truemin,trueval,perComp)
{
  # The agents are column vectors, the last two values are
  # placeholders for average score and standard deviation.
  # We need this competition matrix to include interactions
  nAg <- dim(Agents[[1]])[2]
  modelAg <- cbind(bestAgents[[1]],Agents[[1]])
  lenModelAg <- dim(modelAg)[1]-2
  optAg <- cbind(bestAgents[[2]],Agents[[2]])
  lenOptAg <- dim(optAg)[1]-2
  # Find number of agents
  N <- dim(modelAg)[2]
  compMat <- genCompMat(N,perComp)
  scoreMat <- compMat
  for(i in 1:N)
  {
    for(j in 1:N)
    {
      if(compMat[i,j]==1)
      {
        items <- f(modelAg[1:lenModelAg,i],optAg[1:lenOptAg,j])
        val <- scoreFunc(items,truemin,trueval)
        scoreMat[i,j] <- val
      }
    }
  }
  for(i in 1:N)
  {
    modelVec <- c()
    optVec <- c()
    for(j in 1:N)
    {
      if(compMat[i,j]==1)
      {
        modelVec <- c(modelVec,scoreMat[i,j])
      }
      if(compMat[j,i]==1)
      {
        optVec <- c(optVec,scoreMat[j,i])
      }
    }
    modelAg[lenModelAg+1,i] <- mean(modelVec)
    modelAg[lenModelAg+2,i] <- sd(modelVec)/sqrt(length(modelVec))
    optAg[lenOptAg+1,i] <- mean(optVec)
    optAg[lenOptAg+2,i] <- sd(optVec)/(2*sqrt(length(optVec)))
  }
  #list of best then new
  return(list(list(modelAg[,1:nAg],optAg[,1:nAg]),
      list(modelAg[,(nAg+1):N],optAg[,(nAg+1):N])))
}
```

The function below performs the calculations explained in section 2.4.3.

```
# This function picks agents for the new best population.
# Agents is the new challanging generation, BestAgents
# are the old best agents!
updBestPop <- function(Agents,bestAgents,f,scoreFunc,truemin,trueval)
{
  modelAg <- Agents[[1]]
  lenModelAg <- dim(modelAg)[1]-2
  optAg <- Agents[[2]]
  lenOptAg <- dim(optAg)[1]-2
  # Number of Agents
  N <- dim(modelAg)[2]
  bestModelAg <- bestAgents[[1]]
  bestOptAg <- bestAgents[[2]]
  modelCounter <- 0
  optCounter <- 0
  for(i in 1:N)
  {
    valModelNew <- modelAg[lenModelAg+1,i]-modelAg[lenModelAg+2,i]
    valModelOld <- bestModelAg[lenModelAg+1,i]
    if(valModelNew > valModelOld)
    {
      modelCounter <- modelCounter + 1
      bestModelAg[,i] <- modelAg[,i]
    }
    valOptNew <- optAg[lenOptAg+1,i]+optAg[lenOptAg+2,i]
    valOptOld <- bestOptAg[lenOptAg+1,i]
    if(valOptNew < valOptOld)
    {
      optCounter <- optCounter + 1
      bestOptAg[,i] <- optAg[,i]
    }
  }
  NewAgs <- list(bestModelAg,bestOptAg)
  return(NewAgs)
}
```

The following is the main function we need to call. $f$ is a function which performs the optimization, i.e. the interaction between the predator and prey, while `scoreFunc` summarizes it and calculates the weighted linear combination, see section 2.4.2.

```
# f is the function that returns the parameters for
#   scoreFunc here gadgetCall.
# scoreFunc is the function we would like to minimize.
# truemin and trueval are the parameters at the minimum
#   value of f and its function value, (add if needed)
# numAg is the number of agents in the method
# perComp is a value which should be between one and numAg,
#   best is probably 5-10% of numAg, it controls how much
#   of the competition matrix is filled
# numGen is the number of generations
# lowerf and upperf are bound for the function f
# lowerOpt and upperOpt are bounds for the optimization
#   parameter.
parEst <- function(f,scoreFunc,truemin,trueval,numAg,perComp,numGen,lowerf,upperf,
    lowerOpt, upperOpt, CR, F_par, F_vec)
{
  ######################################################
  #  Creation of initial population                    #
  ######################################################
  # bestAgents is a list with two matrices, Mod and Opt, the last two values of
      each
  # column is the corresponding average score and standard deviation.
  bestAgents <- genAg(numAg,lowerf,upperf, lowerOpt, upperOpt, F_vec)
  ######################################################
  #  Further evolution                                 #
  ######################################################
  HugeList <- list()
  HugeList <- c(HugeList,bestAgents)
  for(i in 1:numGen)
  {
    newAgents <- NewAg(bestAgents,lowerf,upperf, lowerOpt, upperOpt, CR, F_par, F_
        vec)
    allAgents <- calcScore4x4(f,scoreFunc,bestAgents,newAgents,truemin,trueval,
        perComp)
    bestAgents <- allAgents[[1]]
    newAgents <- allAgents[[2]]
    bestAgents<- updBestPop(newAgents,bestAgents,f,scoreFunc,truemin,trueval)
    HugeList <- c(HugeList,bestAgents)
  }
  return(HugeList)
}
```