Optimization

General remarks

We are confronted not only with optimization in mathematics and engineering, but also in almost every aspect of life. Maybe, optimization is most important process of life. When we learn walking, talking, or anything else, our brain is doing some optimization. When we observe natural evolution, we have the impression that some optimization is done. When we look at the development of theories, we can consider this as an optimization process. These observations are fascinating and also inspiring for the development of numerical optimization procedures, especially when difficult, highly complex problems have to be tackled.

In the design of codes for electromagnetics, one can distinguish three different types of optimizations. First of all, such a code should compute the field or some derived quantity in such a way that 'good' results are obtained. When we have defined some error of the result, we look for a code that minimizes the error, which is an optimization procedure. Secondly, we are interested in 'good' codes that are sufficiently accurate, reliable, efficient, and so on. Therefore, code designers try to optimize codes. Instead of an design optimization, one can even try to implement codes that optimize themselves. Such codes exhibit some learning capability. Finally, codes are applied to speed up the design process of electrical engineers who develop new devices, which is again an optimization procedure. The device designer can either do this optimization using his experience and his brain or he can run an optimization procedure on a computer. The latter requires the coupling of a sufficiently reliable and efficient electromagnetics code with an efficient optimization code.

The numerical optimization of code or device design is very demanding, but the quickly growing speed and memory size of modern computers allows us to solve increasing number of increasingly difficult optimization problems within a sufficiently short time. One can foresee that this will cause a revolution in the design process in the near future, because numerical optimization can explore a much wider area of the search space than a human designer. Therefore, numerical optimization is able to 'invent' unconventional and more efficient solutions that clearly outperform conventional solutions of human designers.

The aim of optimization is finding the 'best' or at least 'good' solutions. For computer codes, the goal of the optimization must be precisely defined. In general, a 'cost' or 'fitness' function must be specified. The goal is to minimize the 'cost' or to maximize the 'fitness'. Cost and fitness functions are real valued and may have many variables. The variables represent parameters that characterize the system, code, or device to be optimized. In complex situations it may be hard to define cost or fitness functions because several concurrent properties of the system might be desired. For example, one might want to develop a sensor that is cheap, accurate, and robust. The

desired properties must have an influence on the fitness function, but the definition of the function is not unique and has a big influence on what is considered to be optimal. Moreover, each of the properties depend on some of the design parameters. These dependencies are often quite unclear and complicated. Usual optimization procedures assume that the fitness function is a known and well-defined function of the design parameters.

Design parameters may be complex, real, integer, binary, or other numbers as well as more complicated objects. With an appropriate coding one can map even complicated objects on simple number sets. Therefore, we may consider fitness or cost functions as real functions of one or several parameters that are elements of a number space.

For more information on optimization, see [Polak, 1971], [Press, 1992]. Evolutionary and genetic strategies are discussed in [Bäck, 1996], [Fogel, 1995], [Fröhlich, 1997], [Goldberg, 1989], [Holland, 1975], [Koza, 1992].

Linear optimization

In the previous chapter we have considered series expansions that approximate a given function. These series expansions contain a set of linear parameters that can be computed in such a way that the least squares error of the approximation is minimized. This means that a linear optimization problem is solved. The simplest solution consists of the following steps: 1) Define the goal of the optimization, i.e., the cost function as a real function of the linear parameters. 2) Set the derivatives of these parameters with respect to each of the parameters equal to zero. This leads to a linear system of equations that is characterized by a symmetric matrix. 3) Solve the matrix equation numerically. Note that this method may lead to non-linear systems of equations when the cost function is not defined appropriately. When the cost function is set equal to the square norm of the error, a linear system is obtained. This is the method of least squares.

Note that we have encountered other methods to compute the linear parameters of series expansions. These methods may lead to sub-optimal solutions. Sub-optimal solutions are often acceptable when the computational effort can be reduced. Note that one can have different definitions of cost functions that lead to linear systems of equations. Therefore, a sub-optimal solution can turn out to be optimal when the error definition is modified, but most of the sub-optimal solutions remain sub-optimal for any reasonable error definition. Since the numerical matrix evaluation is always inaccurate, also the optimal solutions are not really optimal in a very strict sense. Despite of this, it makes sense to distinguish between optimal and sub-optimal solutions.

We will see in the following chapters that many of the methods for computational electromagnetics can be written as matrix methods. These methods usually compute the linear parameters either in an optimal or sub-optimal way. However, such methods solve a linear optimization problem.

Linear optimization problems can always be reduced to a matrix equations. There are many numerical algorithms for solving such equations. Note that the design of an appropriate algorithm for solving a given class of matrix equations can also be considered as a highly non-linear optimization process. This process is very demanding. Therefore, one can usually not expect to find an optimal algorithm. Finding at least a sub-optimal matrix solver is important for the design of linear optimization codes because the matrix solver is the most time-consuming part of such codes.

For more information on matrix equations, see the next chapter and textbooks [Arfken, 1985], [Golub, 1983].

Non-linear optimization

Non-linear optimization is much more difficult and usually also much more time-consuming than linear optimization. There is no unique way to solve non-linear optimization problems. Usually, iterative techniques are applied that converge toward the global minimum or at least toward a local minimum of the cost function. Note that finding the minims of cost functions is essentially the same as finding the maxims of fitness functions. For reasons of simplicity, we focus on the cost functions in this section. Finding the global minimum is much more difficult then finding a local minimum – except in situations with a single minimum. As soon as the number of minims is infinite, finding the global minimum is impossible.

Fortunately, finding the global minimum is rarely required in engineering. Usually, it is even sufficient to find an acceptable solution that meets some specifications. This means that one searches for an area where the cost function is below a certain barrier. Thus, it is not even necessary to detect local minims. Despite of this, many of the traditional algorithms converge toward a local minimum. When this minimum is on the above the barrier, one must restart the search. Since the detected minimum depends on the start point of the search, one must modify the start point and hope that the search will find a deeper minimum.

Real parameter optimization

To illustrate the procedure, we can look at the most simple case of a one-dimensional cost function $f(p)$ with a real parameter p defined in a finite interval $p_a \ldots p_b$. To start with, let us assume that the function is very nice in the sense that it can be differentiated infinitely many time and that its derivatives are continuous everywhere. A typical example is shown in Figure 1. Assume that we already have an algorithm that finds a local minimum when it is started anywhere. When the function has more than one minimum, we do not know which of them will be found and when the algorithm has found a minimum, we don't know whether there is another, deeper minimum somewhere else. When we run the same algorithm with another start point, we do not know whether it will find another minimum or the same as before.

Rough search

Therefore, it may be reasonable to start with a rough search that scans the entire interval in a certain number of points. An alternative would be a search that starts at one side of the interval. As soon as this search has found the first minimum, it continues and now searches for a maximum. After detecting a maximum, it searches a minimum again, until it reaches the other end of the interval. An initial rough search allows the user to obtain some graphical insight that may be helpful for selecting an appropriate minimum search routine. The main problem of the rough search is that one should test the function at a sufficiently high number of points for capturing all minims. This is far from being trivial, because some functions have very sharp minims that require an extremely high resolution. Therefore, the rough search does never guarantee that all minims have been detected when one does not have some additional knowledge of the behavior of the function.

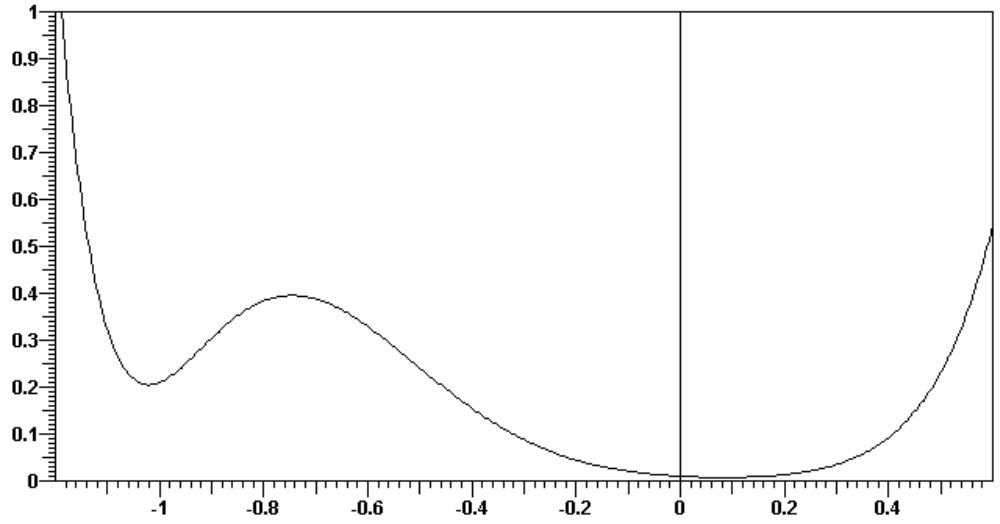

Figure 1: Simple cost function with two local minims.

Beside very sharp minims, a cost function can also have an infinite number of minims. Figure 2 shows a cost function with an infinite number of minims. Despite of this, the global minimum may be detected with an initial rough search, provided that the initial scan is sufficiently fine.

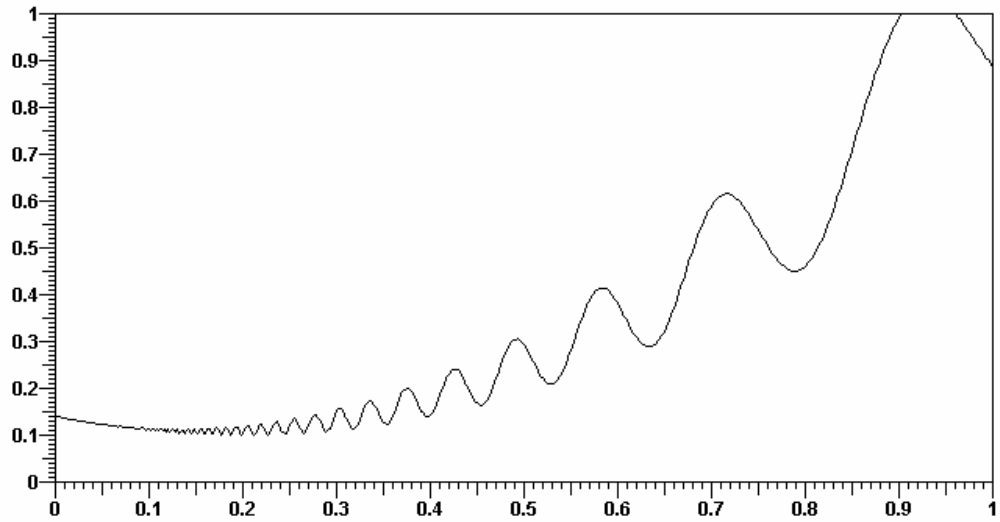

Figure 2: Cost function with infinitely many local minims. Despite of this, the search for the global minimum may be successful.

Figure 3 shows another cost function with infinitely many minims that are so sharp that it is even impossible to show a correct picture. Since the minims are located near the origin, you may 'zoom in'. From this, you may guess that the global minimum is $f(0)=0$. Despite of this, all classical numeric search routines will run into severe problems when you let them analyze this function. Note that even the rough search alone will allow you to find non-minimal solutions below a given barrier.

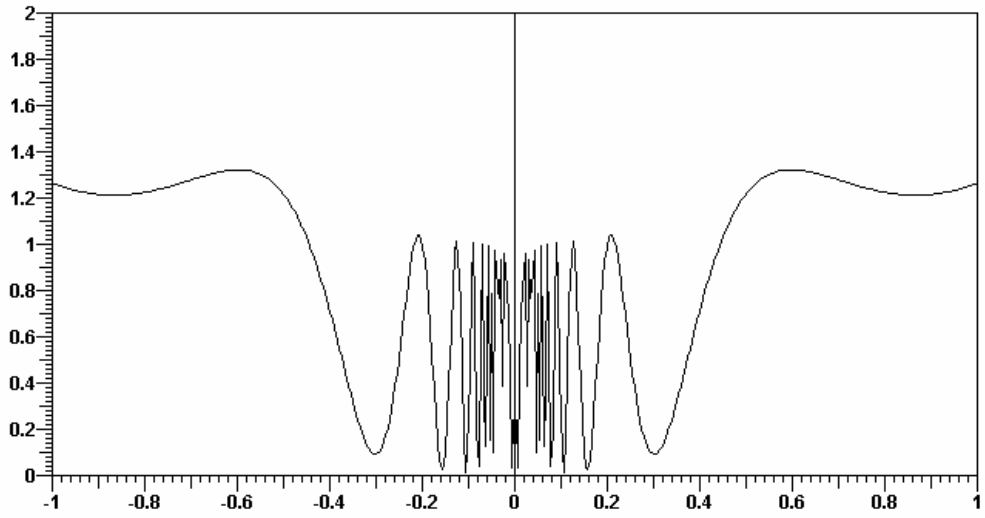

Figure 3: Another cost function with infinitely many local minims. The numeric search for the global minimum will fail.

Obviously, the rough search becomes much more demanding when the number of parameters of the cost functions becomes bigger. When you require N test points for the rough search of each parameters, you require N^m test points for the rough search of a cost function with m parameters. Thus, it is hopeless to perform a rough search when the number of parameters is too high or when the cost function has too sharp minims.

Fine search

After the initial rough search, one usually starts a more or less sophisticated fine search. Even when the rough search cannot be performed (for the reasons mentioned above) one may start the fine search at an arbitrary point in the parameter space and observe what happens. When one searches for values where the cost function is below a given barrier, one will stop the fine search when either such a value has been found or a local minimum has been detected. When the local minimum is above the barrier, one can restart the fine search from another start point.

There are two main classes of fine search algorithms: 1) algorithms that need to know the values of the functions and of their n-th order derivatives and 2) algorithms that work with the function values only. The latter provide additional numerical problems, but they can be applied to more general problems. The most simple algorithms of the first class use the first order derivatives only, i.e., the gradient information of the cost function. When we know that the second order derivatives are continuous, we also know that the gradient of the cost function is zero at each minimum. Therefore, gradient search allows one to replace the minimum search algorithm by a zero search algorithm that is numerically much easier.

To illustrate that even zero search of simple functions is not trivial, let us consider the search for the zeros of the function z^n-1 in the complex plane. You may consider this as the search of the zeros of a function with two real parameters, but working in the complex plane is simpler. The function z^n-1 has n zeros at the positions $e^{i2k\pi/n}$, where k=1,2,…,n. When we start a classical Newton [Press, 1992] algorithm for finding the zeros at a point z_0 of the complex plane, it will find one of the zeros, z_k. Which of the zeros will it find? Probably, you guess that it will find the zero that is closest to the start point z_0. If this would be correct, you could find the complex z_0 plane of the possible start points in n sectors. When starting in the sector number k, you would find the solution number k. It turns out that the complex plane is subdivided into n domains that are fractals [Mandelbrot, 1977] rather than simple sectors. Figure 4 illustrates this for n=5. The

structure obtained is characteristic for the search algorithm that is applied, but exploring this structure is extremely demanding both from the numeric and analytic point of view.

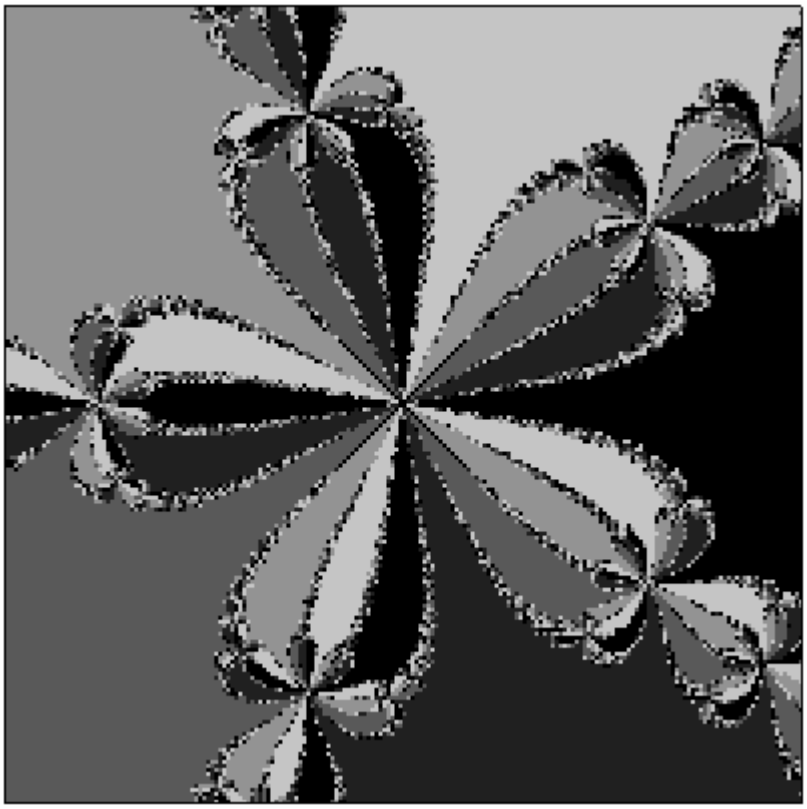

Figure 4: Newton search of the zeros of z^5-1 in the complex plane. The different gray scales indicates the areas of the plane that lead to one and the same zero when the Newton algorithm is started from there. The resulting geometric structure is a fractal.

The Newton search algorithm is a very simple. The zero that is found depends only on the start point. More sophisticated search algorithms contain many user-definable parameters that have a big influence on the search behavior. Analyzing such algorithms is much more difficult and time consuming. Therefore, one often applies search algorithms without precise information on the search pattern. This makes the selection of appropriate algorithms and the development of such algorithms very demanding.

A description of real parameter search algorithms is out of the scope of this book. For more information, consult a textbook, for example, [Press, 1992].

So far, we have only considered relatively 'nice' cost functions. It is obvious that discontinuities and other 'ugly' properties of the cost function cause additional problems that can considerably disturb classical search algorithms. For example, the accuracy of the numerical evaluation of the cost function (and its derivatives) is limited. This causes some disturbing noise. For example, near a zero of the first derivative of a cost function, the noise can cause additional zeros. This can disturb a gradient search algorithm. There are several ways to overcome such problems, but this makes the algorithms more complicated.

Finite parameter sets

One might believe that the minimum search of a cost function with non-real parameters is simpler when the parameter set is finite. This is correct when the parameter set is so small that one can evaluate the cost function for all possible parameters. Unfortunately, the power of computers is limited and the size of parameter sets of typical engineering applications is so big that one can explore only a small part of the entire search space. Even in relatively simple situations the search space can have in the order of 10^{100} points. Moreover, the numerical evaluation of the cost function may be time-consuming. For example, one may have to solve a field computation problem that takes several seconds even on a fast machine. Therefore, one rarely can compute the cost function in more than 10^6 points.

Another difficulty arises from the structure of the search space. An n-dimensional search space with real parameters usually has nice properties. First of all, one can define a metric. This is essential for search algorithms that should know which points of the space are close to a given point and which points are far away. When one talks of 'neighborhood search', this means that the metric of the search space is known. Moreover, one can usually define scalar products. We have seen the benefits of scalar products in the previous chapters. As soon as one works with cost functions with finite parameter sets, one is confronted with non-metric spaces. How can one explore a non-metric space?

The most simple search algorithm that works in any situation is random search. In complicated engineering applications, we do not expect that 'blind' random search is able to find some solution that outperforms the solution of an experienced human designer. Such a solution might be obtained by chance, but a more goal-oriented search should be possible and more promising. We are tempted to say that random search will be far from efficient and that there should be much more efficient, 'intelligent' search procedures. Unfortunately, experience demonstrates that such statements are not correct at all. The efficiency of search procedures depends very much on the problem to be solved. It seems that we are often confronted with problems that can be solved more efficiently than by random search, but there is no search procedure that is efficient for everything. Classical gradient search are most efficient for an extreme class of very nice cost functions, whereas random search is most efficient for another extreme class of very ugly cost functions. In the following, we try finding more intelligent algorithms that can outperform random search in highly complex but not completely random problems.

Random search

Classical non-linear optimization procedures work well when the cost function is 'nice', i.e., sufficiently smooth, differentiable, when it has a small number of minims, etc. Discontinuities at unknown positions in the cost function cause severe problems for such problems. From this point of view, a random cost function is most demanding because it is discontinuous everywhere. Finding the absolute minimum of a random function is completely impossible. Although there is a small chance of hitting it, one never can know whether the minimum has been found or not. Despite of this, one can search for values of the random function that are sufficiently small, i.e., below a certain barrier defined by the user. Finding such values may have no practical interest, but complex optimization problems may have cost functions with many discontinuities at unknown positions. For such problems, non-deterministic algorithms may be more efficient than classical, deterministic search. Random search is the most simple non-deterministic algorithm: A random generator decides at which point of the search space the cost function shall be evaluated. As soon as a value below the given barrier is found, the algorithm can stop.

It can be proven that random search is the best method for finding sufficiently small values of a random function rnd(p). Random search means that one randomly selects the parameter p. What does it mean when we say that random search is the best method? The properties of rnd(p) do not allow one to say that any method is able to find a solution with rnd(p) < e within a finite number of steps, but it is obvious that each method has a certain chance of finding such a solution within a certain number of steps. Therefore, a proper mathematical definition of the quality of a search procedure is quite sophisticated.

Unfortunately, random search is extremely inefficient when we search for minims of more 'brave' functions. For example, to find the minimum of a parabolic function $f(p) = p^2$ in the interval p=0…1, random search would randomly set n points in the given interval. One has a good chance that the procedure would set one point in the interval 0…1/n. To obtain a solution with $f(p) < e$, one can expect that approximately $1/\sqrt{e}$ evaluations of $f(p)$ are required. When e is small, this is a huge number of function calls. Classical search procedures such as gradient search algorithms, algorithms based on parabolic interpolation, etc. could find the minimum within machine precision with a finite number of function calls. In the best case, two function calls and two gradient evaluations would be sufficient – beside algorithms that find the minimum 'by chance' within less steps.

Since classical optimization procedures have severe problems with more complex cost functions, it makes sense to combine random search methods with classical ones. For example, one can let the random procedure set a point within the search interval and explore the neighborhood afterwards by a classical algorithm – provided that some metric is defined. As soon as this algorithm has detected either a local minimum or numeric problems, one can let the random procedure set another point, and so on. Such algorithms are promising when the cost function is piecewise continuous. For more complex situations, other algorithms must be designed, especially, for non-metric search spaces.

Evolutionary Algorithms (EA)

When one observes nature, one obtains the impression that one can see different optimization procedures working on different levels of a highly complex system. The optimization problems 'solved' in nature seem to be much more complex and demanding that difficult engineering problems. Therefore, we may get hints from nature.

The most prominent example is the natural evolution observed in biology. It seems that even simple cells could not have been generated by a random combination of molecules that were available in the atmosphere. In fact, such a statement is not precise enough and even when the chance of a random generation of cells is extremely small, this does not prove that no random generation took place. It is well known that Darwin proposed a theory of evolution that is based on some simple concepts, such as 'natural selection', 'survival of the fittest', etc. There is still an extended discussion on how evolution works and we cannot claim that we really know the mechanisms of evolution. Despite of this, we can implement optimization procedures that are based on simplified models of natural evolution and of other optimizations that seem to be observed in biology, economy, etc.

In order to demonstrate the procedure, the implementation of genetic optimization is outlined in the following. This type of optimization has successfully been applied to symbolic regression and other complicated optimization problems [Fröhlich, 1997], [Quagliarella, 1998], [Winter, 1995].

Genetic Algorithms (GA)

The first Genetic Algorithm (GA) was developed by Holland in 1975 [Holland , 1975]. Since then, GA's have been applied to many problems in various areas of science and engineering [Fröhlich, 1997], [Quagliarella, 1998], [Winter, 1995]. In the following, the GA concept is outlined and some hints are given how more efficient GA codes might be developed. Standard GA's roughly imitate the process of natural evolution. Improved GA's can be created by better imitations of the process observed in nature, by generalizations of these procedures, or by adding new concepts that are not observed in natural evolution.

Inside a computer, everything is represented by a simple bitstring. Therefore, also the solution of any problem must be represented by a bitstring. A GA does nothing else than finding such bitstrings. It plays no role whether we search for a number, a set of numbers, a function, or even an algorithm, we can code this into a bitstring and we can let a GA search for an appropriate bitstring. The GA is not much more than a machine that produces bitstrings of a desired length. The GA user defines the coding and tells the GA the desired length of the bitstrings. When a GA outputs a bitstring, the user has to evaluate the quality of the bitstring and to return a fitness number to the GA. When the user wants to let the GA minimize a cost function, he can define the fitness as the inverse of the cost. The GA will use the fitness evaluation to construct more promising bitstrings.

Although GA's can be applied to any optimization problem, we focus on the problem of symbolic regression to demonstrate the procedure. Note that symbolic regression may include non-linear optimization of real parameters, as well as finite parameter sets. The goal of symbolic regression is finding a formula that is a good approximation of a given formula. To apply a GA, we first have to define the coding.

Coding

Assume that we already have some coding that defines how a formula is mapped on a bitstring. Then, we can let our computer generate bitstrings that describe formulae. This situation is similar to the genetic information that describes an animal. The former is called genotype, the latter is the phenotype. In a first approach, it plays no role for us whether the genotype defines the phenotype exactly and uniquely. We can ignore the fact that there may be differences between different individuals with identical genotype caused by the influence of the environment. When we want to solve a symbolic regression problem with a GA, a bitstring corresponds to the genotype and the corresponding formula corresponds to the phenotype.

Obviously, there are many ways how a formula can be coded into a bitstring and it is also quite obvious that the coding can have an important influence on the GA performance. Assume that we want to code a set of most simple formulae that consist of a single function with two arguments, i.e., we have a tree with one node and two branches. Assume that we have four elementary functions that correspond to the operators +,-,*,/. Moreover, the terminals shall be either the variable x or a constant. First, we decide how to code the constant. Depending on the accuracy and range of this constant, we must reserve a certain number of bits in the string. For example, when the constant is an integer value in the range 0…9, we need at least four bits for coding it, but with four bits we can code more than 10 different integers. Four bits correspond to an integer number in the range 0…15. When the result is bigger than 9, we decide to insert the variable x otherwise, we insert the constant. Since we have two terminals, we need 8 bits. To decide which of the four possible operators should be applied, we need another two bits. Thus, the first two bits could characterize the operator, the bits 3 up to 6 the first terminal, and the bits 7 up to 10 the

second terminal. For example, `1011110011` might be the genotype of the formula x*3. Since x*3 is equal to 3*x, `1000111111` is another genotype with the same phenotype. Also `1000111101` has the same genotype because both `1111`=15 and `1101`=13 correspond to numbers >9 that cause the variable terminal x.

When we generate bitstrings by a random procedure, the coding has an influence on how frequently a certain genotype is generated. In the previous example, one could use one or several extra bits for each terminal. This would considerably increase the probability of having a variable terminal instead of a constant one. Have you an idea how you could code the formula with a reduced probability of having a variable terminal? This is not very difficult. If you wish, you can also code the formula in such a way that the probability of having the operation * is higher than the probability of having another operation. A high probability of a certain phenotype is obtained when many different genotypes have the same phenotype. Note that such a coding with different probabilities leads to longer bitstrings.

Note that the genotype of an animal does not entirely define the phenotype in nature, nutrition, education, etc. can have a strong influence. Such facts are not implemented in standard GA's.

Although coding has a big influence on the GA performance, it is somehow outside the GA business. GA codes generate the bitstrings and not the phenotypes. The coding is the most demanding task for the GA user.

Mutation

How does nature 'design' and 'optimize' animals? First, we can observe that animals create 'children'. The children of simple animals have almost the same genotype as their parents. Typically, each child has one parent and mutation is the mechanism that modifies the genotype. The implementation of bitstring mutation is obvious and simple: One flips one or several bits of the string. For example, for a parent with a bitstring of length 6 and a mutation of the third bit, you might have the following:

```
Parent:010011 → Child:011011
```

Mutation of the third bit

Crossover

After mutation, nature has invented a more complicated, 'sexual' process that modifies the genotype: crossover. Crossover requires at least two parents to create children. Crossover means that the genetic information of the parents is split into (at least) two parts. The child then obtains a new genotype that is a mixture of the genotypes of its parents. It seems that crossover speeds up the progress of evolution. The implementation of crossover mechanisms for bitstrings is also simple. Unlike in nature, one can work with more than two parents. The simplest crossover is obtained when one has two parents with bitstrings of the same length and when one cuts these strings at one and the same position. For example, for two parents with bitstrings of length 6 and crossover after the second bit, you might have the following:

```
Parent1:010011 → Part11:01, Part12:0011 → Child1:=Part11+Part22:011010
Parent2:111010 → Part21:11, Part22:1010 → Child2:=Part11+Part22:110011
```

Crossover after the second bit

You may easily generalize the crossover procedure for more than two parents and more than one crossover points. Note that this does not allow you to create children that cannot be generated (at least indirectly) by one-point crossover with two parents. For example, you may have the following:

```
Parent1:010011 → Part11:01, Part12:00, Part 13:11
Parent2:111010 → Part21:11, Part22:10, Part 23:10
Parent3:001100 → Part31:00, Part32:11, Part 33:00
```

Crossover after the second and fourth bits

One of the children is, for example, `Part11+Part32+Part23:011011`. There is no way to select two of these parents in such a way that the child is directly generated by a one-point crossover, but there are several ways to generate the child within two steps. For example, with a crossover after the first bit you can generate a first child with the bitstring `011010` by mating parent1 and parent2. Now, you can combine this child again with parent1, crossover after bit 5 for obtaining `011011`. Obviously, you might have created this also with a two-point crossover of the first two parents.

Randomness

Multi-parent crossover and multi-point crossover allow one to skip intermediate steps required by one-point crossover with two parents to reach a certain bitstring from a given population of parents. Thus, one might reach optimal bitstrings more quickly. At the same time, the chance of generating much worse individuals is also increased. Therefore, it is not obvious how many parents and how many crossover points should be admitted. The more parents and the more crossover points one has, the bigger the jumps in the search space of the bitstrings caused by crossover. Is this desired? Note that one could also reach every possible bitstring from any bitstring by multi-point mutation. Such 'far' jumps therefore are similar to a random search. There is no reason for a Genetic Algorithm (GA) that does not outperform random search. Therefore, we should be careful when we introduce generalization such as multi-parent, multi-point crossover. Note that a proper definition of terms like 'near' and 'far' would require the definition of a metric, which is not always possible. Therefore, precise formulations can be tedious.

Remember that it is impossible to outperform random search in extreme situations, such as finding minims of a random function. Depending on the problem to be solved, it may be reasonable to push the GA search toward random search, but in most of the engineering applications the randomness inherent in simple GA search is already sufficiently high or even too high.

The most simple way to control the randomness of a GA is a combination of crossover with mutation. After crossover one can apply mutation to the children with a certain rate. In most of the practical applications, it turns out that the mutation rate should be low, but not equal to zero. The optimal mutation rate depends on the problem to be solved and on the GA implementation. For so-called steady-state GA a mutation rate of approximately 0.1 is appropriate, whereas lower mutation rates are optimal for other implementations.

Population

The individuals of a GA are members of a population. Working with big populations means using much memory. The limited population size in nature as well as in GA implementations implies that individuals must die. When an individual dies, the corresponding information and experience is lost. Although this is a drawback at first sight, it also avoids cumulating of information contained in the population, which would cause problems in the selection process (see below). In nature, one can observe that smaller populations may be more flexible, whereas large populations can become more stable or even conservative. To perform crossover, at least two parents must be available, i.e., the minimum population size is two. It is quite obvious that a GA with such a minimal population is not optimal. Finding an optimal population size is not easy at all. This requires much experience and depends on the complexity of the problem to be solved and on the specific GA implementation. In engineering problems, the fitness evaluation required for each individual may be time-consuming. In this case, one is forced to work with relatively small populations of about 100 individuals, even if larger populations would improve the GA performance.

Instead of working with populations of fixed size, one can also work with populations of varying size, which is also observed in nature. This is especially promising when the fitness evaluation is time-consuming. In this case, it seems to be natural to start with a small population and to let the population grow until a certain limit is reached. However, classical GA work with fixed size populations.

Initialization

When a GA is started, one usually assumes that no information is available. Therefore, all well-known GA implementations start with a pure random initialization of the population. The initial population is called generation number 0. With a random initialization, one obtains high randomness of the GA search when one works with a big initial population, whereas GA with smaller populations are further away from random search.

Note that non-random initializations might be attractive in practice. For example, one might start with a user-defined initialization. This is not easy because the user certainly would hate to define initial bitstrings. Instead of this, he might be able to 'propose' some initial phenotypes that are promising. Therefore, user-defined initialization requires a coding of given phenotypes into bitstrings.

In general, one can consider a GA as an iterative process. Any iterative process can be supported by an appropriate initialization that takes prior knowledge into account. This knowledge may be obtained from analytical considerations, from previous computations, from known solutions of problems similar to the one to be solved, from practical experience, and so on.

Generations

Starting with the initial generation 0, a GA creates a new generation using the operators copy, crossover, and mutation applied to the parents. This procedure is repeated iteratively.

Most GA implementations work with generations with a fixed population size. This means that the size of generation n is the same as the size of generation n-1. Because of the copy operator, some individuals of generation n-1 are also contained in generation n, but others 'die'.

Note that one has neither a constant population size nor a strict separation of generations in nature. Moreover, the lifetime of different individuals can be very different. Therefore, standard

GA implementations mimic natural procedures very roughly. Since we cannot say that natural optimization procedure is optimal, this does not mean that standard GA must perform badly.

Elitism

One of the most important drawbacks of the usual fixed-size generation implementation is that even the best individuals of a generation can die when the next generation is created. As a consequence, the best individual of the new generation may even be worse than the best one that had found before. To avoid this, one can always keep the best (or the *m* best) individuals of a generation by copying the corresponding bitstrings into the new generation. This is called elitism.

Completeness

When a new generation is created, it is hoped that the individuals of this generation concentrate more on those areas of the search space where the individuals of the previous generation were successful. Like in classical minimum search algorithms, there is a certain chance that a GA is 'trapped' by some local minims. Because of the randomness contained in the GA mutations, escaping the trap is always possible, but in this case, the GA becomes even slower than random search. In this case, special procedures might help the GA to escape the trap. How can one detect that a GA is trapped? Mutations allow the GA to escape a trap in a relatively inefficient way. Moreover, the mutation rate is usually much smaller than the crossover rate. Therefore, we best focus on crossover.

First of all, it is possible to create all possible bitstrings of a finite length from two parents only, provided that the corresponding two bitstrings are complementary, i.e., when the logic XOR operation of the two bitstrings results in a bitstring of the form `111111`. When one has a population of $N > 1$ different parents, it is always possible to generate an arbitrary bitstring by crossover when one has at least one `0` and one `1` bit in at least one of the parents for each of the positions of the bitstrings. Therefore, it is relatively simple to make sure that crossover can generate all possible bitstrings, i.e., that it can theoretically explore the entire search space. Unfortunately, this does not guarantee a quick escape from traps.

Diversity

To get more information on a generation, a statistical analysis of the bitstrings within a generation may be helpful. This analysis should clarify how diverse the individuals are. Obviously, there are many different ways to define the diversity. Without a precise definition of the diversity, we can guess that it might be good when one has a similar number of `0`'s and `1`'s in each bit position of all individuals. For more precise definitions of the diversity see [Fröhlich, 1997].

To achieve a sufficiently diverse generation, one might first use crossover and mutation for creating a part of the generation only. Afterwards, one can fill the rest of the generation with individuals that increase the diversity.

Fitness

From observations we know that mutations and crossover provide a certain chance of generating a 'better' child that its fitter than its parent, but also the danger that the child is not able to survive. The terms 'good', 'bad', 'better', 'best', etc. are vague and need a proper definition that permits a GA implementation.

In biology, it is not clear at all what these terms mean. The principle of the 'survival of the fittest' replaces this term by another one that seems to be more scientific and precise, but the principle does not define the term 'fitness' at all. When we observe animals, we are usually unable to say which of them is the fittest. Thus, we are unable to apply the principle of the 'survival of the fittest' to foresee which of the animals will not survive, which of them will have many children, and so on. When we observe that an animal dies in a specific situation, we might be tempted to say that is was less fit than another one that survived. But in another situation, it might have survived, whereas the other one would have died. However, a simple fitness definition could be obtained from the number of children produced by an individual. Would you agree with the statement that mice are fitter than humans because they have more children in the average? Maybe, you want to restrict such a fitness definition to one species only, but would you agree with the statement that people without children have a zero fitness? What about the fitness of an individual with many children when all of the children are sterile? When we look at bees, we see that even sterile individuals can have a big value for a society. When looking at birds, we find birds that can fly and swim very well and we find other birds that can neither fly nor swim, but they still survive. What is the influence of some specific skills on the fitness of an individual? It seems that the environment has a big influence on the fitness of animals. The ability to fly may be useful within a certain environment but useless in another one. Thus, the fitness does not only depend on the properties of an individual. Finally, it is even questionable whether attaching a fitness value to each individual is reasonable.

The dependence of the fitness on the environment is much stronger than one might expect. For example, when we decide to have a nice bird at home and catch a nice one, attributes that were important for the survival of the bird in the nature may be come unimportant and vice versa. Breeding of animals may modify their skills within a few generations in such a way that the animal might even be unable to survive without human support. Breeding means that we redefine the fitness of an animal in a very specific way and that we impose a selection mechanism that is much more rigorous that what we observe in nature. Obviously, breeding is extremely successful in some sense. It speeds up the development of some animal in a direction we have specified by deciding what we consider to be nice attributes of the animal. Of course, we cannot expect that breeding can create a flying cat that could catch birds more efficiently although we think that such an animal might be possible. However, breeding really works within some limitations.

In computer optimization, one typically has a mathematical formalism that allows one to define the quality of an individual much more precisely, but the fitness definition is never unique and it has a considerable impact on the GA performance. For example, when a GA is applied to symbolic regression of a given function $f(x)$, each bitstring represents a function $f_n(x)$ that is an approximation of $f(x)$. In the previous sections we usually applied the square norm of the difference $f(x)-f_n(x)$ to define an error number e_n. The square norm was important because it usually led to a linear system of equations. Now, we can use any norm because we do not even intend to obtain a linear system of equations. Since small errors are desired, the fitness definition $fit_n=1/e_n$ seems to be reasonable, but this is also not unique. If you wish, you can apply some fitness filter of the form $fit_n=Filter(e_n)$. In this case, *Filter* should be a continuous function that has a maximum when the argument is zero.

Since the fitness is obtained from an analysis of the phenotype, the fitness definition and the fitness evaluation are also outside the GA itself. The GA user must not only provide the coding of the genotype into the phenotype, but also the computation of the fitness for a given bitstring. A GA does not much more than producing bitstrings. It needs the fitness of all individuals of a generation for deciding how to create the next generation.

Selection

Assume that the GA has created a generation. It then passes the bitstrings of all individuals to an external routine provided by the GA user. This routine computes and returns the fitness values of all individuals to the GA. Now, the GA creates the next generation using crossover and mutation. For deciding which of the individuals of a generation should become parents, the GA uses the principle of the 'survival of the fittest', i.e., it will more frequently select individuals with higher fitness than individuals with lower fitness. There are several ways to perform this selection. Fitness proportional selection and tournament selection are most frequently applied, but other selection mechanisms could also be designed. Note that a strict selection should be avoided giving 'poor' individuals at least some chance is important for obtaining a good performance. Obviously, fitness filters have a big impact on the selection.

Once the GA has selected parents, it applies crossover to create children. Crossover requires the specification of the crossover point. This point can be randomly selected. Of course, one could use some statistics on the success of the crossover points. This would allow one to attach another type of fitness values to the crossover points and to select the fittest crossover points more frequently than other ones. There are certainly much more sophisticated routines for selecting crossover points, but such routines considered here. A prominent example is so-called uniform crossover.

Remember that the GA will apply mutations with a certain rate to all children of the new generation. If an elitist strategy is implemented, it can copy the best individuals of the previous generation and if desired, it can create special individuals that increase the diversity of the new generation.

Termination

A GA produces one population after another. This can be done in an infinite loop. When the user specifies a fitness value to be reached, the procedure can be stopped as soon as at least one individual has a higher fitness than the desired one.

Often, the user does not exactly know how big the fitness value of an acceptable solution should be. Therefore, one would like to stop a GA when one cannot expect to obtain much better solutions anymore. From the experience with traditional optimization algorithms one might be tempted do observe convergence and to stop the GA when the maximum fitness remains more or less constant over some populations. This should not be done because the GA search exhibits a pronounced staircase behavior. The maximum fitness may be almost constant over many generations and then it can be suddenly increased drastically within a few additional generations. Incidentally, it seems that the same staircase behavior can be observed in natural evolution. Note that the GA staircase is rather a random staircase than a regular one. It is impossible to predict the next step of this staircase. Therefore, an analysis of the GA convergence is useless for obtaining an appropriate stopping criterion.

In most GA applications, the user has some ideas about the desired performance of the object to be optimized by the GA, i.e., the user knows the desired properties of the genotypes that correspond to the bitstrings produced by the GA. In this case, the GA can output the best bitstrings of a generation and the user may study the corresponding phenotypes. As soon as he is convinced that one of the phenotypes is sufficiently good, the user can stop the GA.

Conventional GA structure

The following is a pseudo code of a simple conventional GA. Many of the GA codes that are available for free are mainly based on this structure.

```
I=0
Initialize Population P(I)
Do {
  Evaluate the fitness values of population P(I)
  Stop if terminal condition is met
  While (next population P(I+1) is not full) {
    Select two individuals Ind1, Ind2 out of P(I)
    Copy(Ind1,Ind2) => Offspring1,Offspring2
    If(random<c-rate) Crossover(Ind1,Ind2) => Offspring1,Offspring2
    If(random<m-rate) Mutation(Offspring1) => Offspring1
    If(random<m-rate) Mutation(Offspring2) => Offspring2
    Copy Offsprings into the next population P(I+1)
  }
}
```

Figure 5: Pseudo code of a simple, conventional GA. Note that the crossover and the mutation operations are performed with some probabilities depending on the crossover rate and mutation rate specified by the user. When the crossover is not done and when no mutations are applied, the offsprings are simple copies of the parents.

Note that some randomness affects most of the steps of this GA and that the implementation of these steps is not defined in detail. Therefore, the same pseudo code can be used for different implementations with different behavior. For example, the routine Crossover can randomly select the crossover points. When crossover points before the first bit or after the last bit of the strings are permitted, Crossover also can generate simple copies of the parents. Similarly, the routine Mutation may select the locations of the bits to be flipped. If this location is allowed to be outside the strings, pure copies of the parents can be obtained. This automatically leads to weak form of elitism because the fittest individuals are selected most frequently. Therefore, the fittest individual may have a good chance to be reproduced. To implement a strict elitism, the pseudo code must be modified. In addition to elitism, we have outlined several ideas that can be implemented in the GA to improve its performance. When you analyze these ideas and the GA structure, you will see that some ideas may be embedded without any modification of the main GA structures.

Hybrid GA

When a real parameter is optimized by a GA, it is coded into a bitstring. Now, when mutation is applied to one of the bits, this can either have a small or a big influence on the size of the corresponding parameter, depending on the position of the bit. For example, you may code a real number as follows: 1) The first bit specifies the sign. 2) The bits 2 up to 5 specify the mantissa. 3) The bit 6 specifies the sign of the exponent. 4) The bits 7 up to 8 specify the absolute value of the exponent. Assume that a given bitstring contains the number –12E+3, i.e., the bitstring is 11100011. When the mutation flips bit number 5, one has 11101011, which corresponds to –13E+3, i.e. a number similar to the original one, whereas flipping bit 6 would generate the number –12E-3, which is very different. This means that the same genetic operation will sometimes correspond to little modifications of the phenotype and sometimes to huge modifications. The same happens with crossover. The main reason for this behavior is the fact that the GA itself has no information on the meaning of the different bits of the string.

To achieve a better neighborhood search of real parameters to be optimized by a GA, one can invent several extensions of the conventional GA. Instead of coding a real parameters into the

bitstring, one can code a placeholder and apply a traditional parameter optimization to it. For example, assume that a GA shall solve a symbolic regression problem with real parameters to be optimized. For reasons of simplicity, try a simple tree with one node, two terminals, four operators +,-,*,/. The terminals may either be the variable or a parameter to be optimized. Instead of coding the numeric value of the terminal, you can use a single bit for each terminal. When the bit is zero, the terminal is the variable x, when it is one, the terminal is a parameter p_i to be optimized externally where i is the number of the terminal. For example, the bitstring `1110` might represent the formula p_1*x. The routine that evaluates the fitness of this formula would have to apply some non-linear parameter optimize to evaluate the parameter and the maximum fitness to be reached with this formula. If you whish, you can interpret the individual p_1*x as an individual that is able to 'learn'. The external parameter optimization then corresponds to some learning process.

Non-binary GA

Often, the coding of finite parameter sets into binary strings is not trivial. For example, assume that you have one parameter that can have three states, i.e., three different values. This might happen in the symbolic regression problem when you admit three operators +,-,* only. Direct binary coding of this parameter requires at least two bits. When you reserve two bits in the GA bitstring for this parameter, you obtain four states. What can you do with the fourth extra state?

The simplest way to overcome coding problems consists in the design of non-binary GA. This is a very simple generalization of binary GA. One now has strings of a finite number of elements. Each element can have a finite number of states. The number of states can be one and the same for all elements or it can differ from element to element. The former is simpler, but less general and less flexible.

The combination of non-binary GA with traditional optimization procedures is essentially the same as the combination of binary GA with traditional optimization procedures. For complex optimization problems, such hybrid non-binary GA implementations are most promising.

Genetic Programming (GP)

Although we know that computers work with bitstrings, no computer user likes input and output in the form of bitstrings. Several computer languages have been designed to make communication with computers more easy. These languages use symbols. Moreover, computer codes are subdivided into subroutines that do a meaningful part of the job. Such a subroutine might be used for the evaluation of a specific mathematical function. For example, assume that the function $\sin(x)$ is needed, but not contained in the pool of intrinsic functions of the computer language. Now, you can write a subroutine that evaluates $\sin(x)$ with in a certain range of the argument x with a sufficient precision. To do this, you start with some mathematical literature and you find some series expansion, for example, power series. When you truncate such a series, you obtain an approximate formula that contains the four arithmetic operators +,-,*,/, the variable x, and several constants. For example, $\exp(x) \approx 1 + x + x*x/2 + x*x*x/6$. This formula might have been obtained as a solution of the symbolic regression problem obtained by an appropriate GA. However, humans who search for such a formula will avoid bitstring coding and use algebra instead. Maybe, they will even apply some symbolic mathematics code such as Maple [Char, 1991] or Mathematica [Wolfram, 1991]. Symbolic math codes demonstrate that one can design codes that handle formulae more or less directly. Therefore, one also can write computer codes

for symbolic optimization that design formulae directly instead of coding them into bitstrings. When we apply the genetic concept, this means that the genotype of such codes is a formula instead of a bitstring. Such formulae can easily be represented by trees as at the end of the previous chapter.

Conventional GP structure

Genetic Programming (GP) [Koza, 1992] is an optimization procedure working on symbolic formulae using essentially the same procedure and genetic manipulations as GA. Consequently, the structure of conventional GP is the same as the structure of conventional GA (see Figure 5). All that has to be done to obtain such a GP implementation is to define the individuals and the genetic operations (crossover and mutation) that shall act on them. The fitness evaluation, selection process, etc. is the same as for conventional GA. Therefore, you easily get hints for generalizations when you study the sections on GA.

Tree representation

To illustrate the tree representation of two individuals, let us consider two simple formulae, for example, x-x*x*x/2 and 1+x*x (see Figure 6).

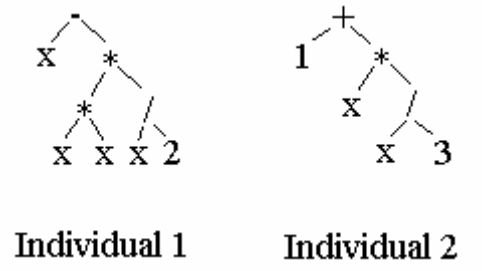

*Figure 6: Tree representations of two formulae, Individual 1 = x-x*x*x/2 and Individual 2 = 1+x*x/3.*

Note that these two individuals have a different topology, i.e., a different number of nodes and a different number of terminals. The depth of both trees is the same, i.e., each of the trees has three different levels of operators, i.e., nodes. In general, the depths of different trees can be different. Because of limited storage capabilities, the maximum tree depth must be limited. This is an important difference between standard GP and GA. The size of the GA bitstrings is typically one and the same for all individuals. Note that also the size of chromosomes is constant for those animals that can be crossed. However, in computers, crossover of individuals with completely different genotypes can easily be defined.

Terminals

Standard GP implementations work with two types of terminals: variables and random constants. With real random constants it is very unlikely to obtain integer values as in the two formulae of the example above. Thus, it is also unlikely to obtain accurate approximations of truncated series expansions such as $\exp(x) \approx 1 + x + x*x/2 + x*x*x/6$. In general, it is expected that appropriate parameter values may be constructed by a branch of the tree with random constant terminals. A typical construction of an approximation of the constant 2 in the formula above with a branch with three random terminals might be (0.783+0.154)/0.912. With sufficiently big trees and sufficiently many random number terminals, one has some chance to approximate a certain

constant. When the GP procedure is smart enough, one can hope that it is able to find a sufficiently accurate solutions. However, this kind of parameter optimization is extremely inefficient.

The Generalized Genetic Programming concept that is outlined in the following section, avoids the difficulties of appropriate constructions of constants or parameters by adding other types of terminals.

Nodes and elementary functions

To each node of a GP formula tree, some elementary operator or function is attached. Modern programming languages know many of them and mathematicians know even more. When one implements a GP code, one has to provide a certain set of elementary functions that are available. The bigger this set is, the more difficult the implementation and the bigger the search space become.

Somehow, the symbolic regression problem is the same as the problem of a scientist who analyzes some measured data and tries to find a simple and useful formula to explain the measurement. Mathematics is often considered as a pure, abstract formalism without any intrinsic correlation to physics or even engineering. When one considers the history of mathematics, physics, and engineering, one observes that some co-evolution took place. Therefore, the best-known mathematical tools are best suited to analyze most of the engineering problems. For example, well-known arithmetic operations and mathematical functions (sin, cos, exp, log) are often very successful to describe engineering problems. The frequent use of a specific function in a special class of problems makes it likely that this function is also involved in a successful solution of a new problem of the same class. The frequent use of certain functions in the solution of engineering problems makes it reasonable to teach such functions to engineers. Consequently, engineers will first try using such functions for solving everyday problems. This is some sort of selection mechanism on mathematical functions that took place in the past. Consequently, an experienced engineer who wants to apply a GP code for solving one of his problems can associate some probability to each of the elementary functions provided by the code. When these probabilities are used for the initialization and mutation of the GP individuals, the efficiency of the GP may be increased.

Mutation

Mutation affects one branch of the genotype. This may be either a node or a terminal. Consider the terminal mutation first: One randomly selects one the terminals. Then one randomly specifies its new type. If its type is a random constant, one can replace it by another random number (see Figure 7).

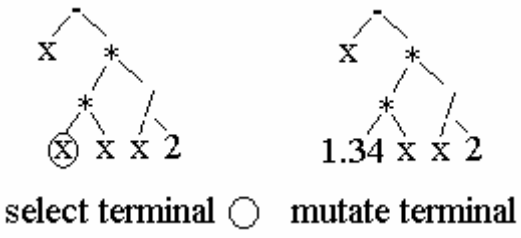

Figure 7: Terminal mutation.

For node mutation, one can randomly select a node and replace the corresponding branch by another one that is randomly constructed (see Figure 8). This mutation is very drastic, especially

when the selected node is far away from the terminal. In this case, there is not much difference between a node mutation and a pure random generation of the individual. Therefore, mutations of standard GP implementations are not very helpful. For this reason, the mutation rate in standard GP is often set to a very low value.

Drastic effects of node mutations may be reduced by mini-mutations that exchange the operator of a node rather than the entire branch. When all nodes correspond to operators or functions with one and the same number of arguments, the mini mutation is easy.

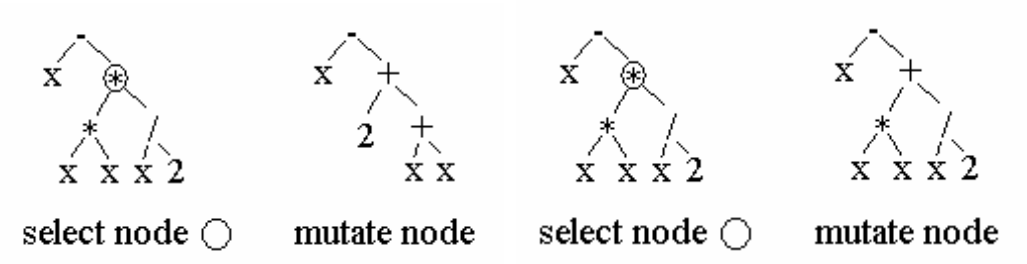

Figure 8: Node mutations. Left hand side: replacement of the node with the entire branch, right hand side: mini mutation that replaces the operator of the node only.

Replacing an operator by another one with a different number of arguments is more difficult because it requires a modification of the tree topology.

Crossover

Since GP works with trees of variable size and architecture, one has to select two crossover points, one for each parent. Each parent is then subdivided into two parts, head and body or branch and trunk. By exchanging the two parts one obtains two offsprings, a child with head of parent one and body of parent two and another child with head of parent two and body of parent one. Figure 9 illustrates this.

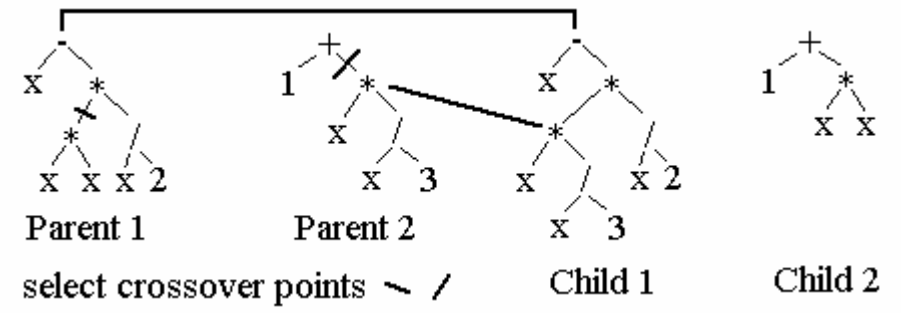

Figure 9: Crossover of two parents for generating two offsprings.

Problems of conventional genetic programming

When one analyzes the performance of conventional genetic programming codes, one can detect many more or less severe problems that make these codes very inefficient even for simple symbolic regression problems. Symbolic regression problems that are often presented for demonstrating the power of GP are well tailored for GP and of little practical value. Therefore, GP codes are often very frustrating for engineers.

It has already been mentioned that conventional GP's try to approximate parameters by more or less long and complicated branches of the formula trees. This kind of parameter optimization is

neither intelligent nor efficient. Maybe, it is the main source of the GP problems in engineering applications.

Another problem that is frequently observed is the fact that after a few generations some of the elementary functions die out, i.e., are no longer contained in any of the individuals. This may also happen to those elementary functions that are essential for a good solution. A typical example of symbolic regression is the analysis of functions of the type $\sin(x)/x$. For small arguments, such functions are similar to cosine. Moreover $\cos(x)$ and $\sin(x)/x$ have the same symmetry. Therefore, it may happen that the GP code throws the sinus function away and favors the cosine. Since the operation / causes numerical problems for small x, it may also happen that this operation dies out. When a mutation generates a new individual that contains / and sin() again, it is very likely that the corresponding tree has a wrong architecture and that the individual has a very low fitness.

Already the simple function $\sin(3.14*x)/x$ combines the two problems mentioned above. When a conventional GP code is used for finding an approximation of this function in the interval x=0…1, one observes that many big populations are required to find solutions that are not too far away from the correct solution. To support the GP, one may reduce the set of elementary functions. Even when one only uses the elementary operations +,-,*,/ and the elementary functions sin() and cos(), the GP turns out to be extremely inefficient. In the following section, some GP extensions and new concepts are proposed that make the procedure much more promising. These extensions illustrate how analysis and generalization of a given method may lead to more powerful methods.

Generalized Genetic Programming (GGP)

The generalization of GP codes by admitting non-linear parameters as a third terminal type corresponds to the hybrid GA combination with a non-linear parameter optimization that has already been proposed in the Hybrid GA section. From this, one can expect a much more efficient handling of real parameters. Although the idea of **parameter terminals** is very simple, its implementation requires considerable modifications of the GP code.

First of all, for the selection mechanisms of GP, a fitness value must be associated to each individual, i.e., formula. Without parameter terminals, the fitness value can be defined as a simple function of the sum of the square errors in the observation or sampling points of the given function. Note that the fitness definition can have a considerable influence on the performance of the GP code, but plausible definitions can easily be found and turn out to be useful in most cases. When parameters are involved, the fitness value of each individual becomes a function of these parameters. It seems to be natural to perform a parameter optimization for each individual and to use the error of the optimal parameter set to compute the fitness value. But non-linear parameter optimizations can be quite time consuming.

Parameter terminals increase the freedom of each individual. They allow the individual to ‘learn’ by adapting the parameters. Although this is a very primitive implementation of **learning**, it can be very useful and efficient and it is a step closer to natural optimization processes that traditional GP. In nature, the ability to learn is certainly important. An individual that learns quickly is probably fitter that another one that learns slowly. The learning characteristics of GGP individuals are connected to the influence of the parameter terminals on the resulting function. Some parameter terminals may have almost no influence on the genotype, whereas other ones can have a big influence. It has been found that very simple and rough non-linear parameter search algorithms combined with GP are sufficient. GGP works with a **symbolic definition of the non-linear parameter optimization**. I.e., the parameter optimization is described by a formula.

Theoretically, an additional GP code could optimize this formula. Since this would be extremely time-consuming, it has not been done yet. The given default formula is simple and has been found to be sufficient in most cases. However, experienced GGP users can easily redefine it.

We have seen that **linear parameter optimization** is numerically much faster and simpler that non-linear parameter optimization. To obtain a set of linear parameters, a series expansion is required. GGP works with a generalized series expansion of the form

$$f(x) \approx \sum_{k=1}^{K} a_k f_k(p_1, p_2, \ldots, p_L, x). \tag{1.1}$$

GGP optimizes the linear parameters a_k, the non-linear parameters p_l, and the shape of the basis functions f_k. This means that a combination of linear parameter optimization, non-linear parameter optimization, and symbolic basis function optimization is applied. For the latter, an advanced GP algorithm is used. When the number of basis functions K is equal to 1, one almost obtains a more simple combination of GP with non-linear parameter optimization only, but there is a little difference: GGP multiplies the basis function with an **amplitude**. Although adding this amplitude is no big deal, it already increases the efficiency of GGP because the construction of the formula can now be focused on the shape of the corresponding genotype, i.e., function. Assume that you have obtained the given function from measurements, then the units that have been used affect the numerical values of the function. When the units are modified, the values are scaled by a certain factor. This will only affect the amplitude, provided that the basis function is multiplied with an amplitude. A traditional GP would have to construct an entirely new formula when the function is scaled with any factor. GGP only has to modify the amplitude, but it can keep one and the same basis.

As soon as a true series expansion with $K > 1$ is used, the implementation becomes much more complex and tricky, although the linear parameter optimization leads to a matrix equation that can be solved easily. The main problem is caused by the fitness definition. One now works with a group of individuals rather than with a single individual. From the error of the approximation of the given function, one therefore can define a fitness value for the entire group, but not for the individuals. Since this causes a quite complex extension, we postpone this generalization and focus on the GGP implementation with K=1. However, it should be mentioned that GGP also uses **symbolic fitness definitions**. Therefore, also the fitness definition formulae could be modified by experienced users or optimized by an additional GP algorithm.

It has already been mentioned that the extrapolation is much more difficult than the approximation of the given function. The same statement holds when one considers the construction of a theory. It is much simpler to explain known experiments than predicting how new experiments will turn out. The main benefit of good theories is the prediction of new and surprising results. Symbolic regression is a simple prototype of the theory building process. In order to obtain good extrapolations, GGP splits the known data set, i.e., the interval where the given function is known, into two parts. The first part is used for computing the basis functions, optimizing the parameter set, and for computing the error of the approximation, whereas the second one is only used for checking the quality of the extrapolation. The GGP fitness is obtained from an analysis of both approximation and **extrapolation error**.

In order to avoid time-consuming non-linear parameter optimizations, GGP performs some simple **initial checks**. Individuals that do not pass, are '**repair**ed' by simple '**mini mutations**'. Repaired individuals that still do not pass the initial test, are discarded without any fitness computation. Trying to repair an individual is a typical idea of an engineer. It seems that introns play a similar role in nature. However, repairing with mini mutations corresponds to an improved neighborhood search and seems to be useful. Mini mutations are simple operations that leave the

structure of the individual intact. The exchange of a constant terminal with a parameter terminal is a typical mini mutation.

Random constants that are used in standard GP implementations have usually nothing to do with physical characteristics of the observed object, but the user often has some idea on constants that might play an important role. Therefore, GGP replaces the random constants by **user-definable constants**.

Mini mutations allow GGP to convert optimized parameter values into a second type of **explicit constants**. This offers another interesting possibility. Only one non-linear parameters per individual is required. This means that (1.1) is replaced by the more simple expression

$$f(x) \approx \sum_{k=1}^{K} a_k f_k(p_k, x). \tag{1.2}$$

From time to time, the non-linear parameter is 'frozen' into a constant and a constant becomes a parameter that can be optimized. The reduction to a single non-linear parameter considerably reduces the iterations required for the non-linear parameter optimization. GGP performs some **re-evaluations of the fittest** individuals because it also uses an elitist strategy. Consequently, a very rough parameter optimization with only 2-6 iterations is sufficient in most situations.

The biggest part of the GGP code has been written in Fortran. Old-fashioned Fortran programmers allocate memory for fixed-length character strings that will contain the formula of the individuals. If they are used to work on small computers, they hate allocating large strings knowing that a big part of them will not be used. Thus, they would prefer a fixed formula size of the individuals. Beside this, there are also 'biological' reasons, not only for a fixed formula size, but also for a fixed formula topology of all individuals. GGP can handle both, **variable and fixed size individuals**. Remember that the structure of a formula composed of elementary functions and operations can be represented by a tree consisting of nodes representing the elementary functions and operations and by branches representing their arguments. The most simple structure of a tree with fixed size and structure is obtained, when all elementary functions and operations have two arguments. Let us call this a **binary tree**. A binary tree of depth 1 has one node and two terminals. A **complete binary tree** of depth 2 has 1+2=3 nodes and 2*2=4 terminals, and so on. Incomplete trees of a given depth have less nodes and less terminals than complete trees. For each depth there is only one complete binary tree. The restriction to complete trees of a fixed depth considerably reduces the search space, which can help increasing the search speed. Obviously, this can also cause problems. First of all, we have to find appropriate replacements for elementary functions with only one argument. This is easy. In many cases, we can replace functions with one argument by generalizations, for example, the elementary function $\exp(b)=e^b$ can be replaced by the more general function $\mathrm{pow}(a,b):=a^b$. Instead, we can also introduce an arithmetic operation in the argument, for example, by defining $\mathrm{com}(a,b):=\cos(a*b)$, and so on. In addition to these **generalizations of functions** with one argument, we need a new type of elementary functions that simply pass one of the arguments and are called **transfer functions**. These functions are $\mathrm{ta1}(a,b):=a$ and $\mathrm{ta2}(a,b):=b$. Transfer functions make one branch of the tree inactive. The phenotype of a complete binary tree containing transfer functions is identical with the phenotype of all trees obtained from mutations in the inactive branches. This means that the information of the inactive branches is hidden and has no influence on the phenotype. Despite of this, the information is still there and can become active in another generation. It seems that this corresponds to so-called **introns** that are also observed in nature.

The default set of GGP functions contains four arithmetic operations, four elementary functions with two arguments, and two transfer functions: $\mathrm{add}(a,b):=a+b$, $\mathrm{mul}(a,b):=a*b$, $\mathrm{sub}(a,b):=a-b$, $\mathrm{div}(a,b):=a/b$, $\mathrm{com}(a,b):=\cos(a*b)$, $\mathrm{sim}(a,b):=\sin(a*b)$, $\mathrm{pow}(a,b):=a^b$, $\mathrm{lga}(a,b):=\log_a(b)$,

ta1(a,b):=a, ta2(a,b):=b. In addition, GGP knows other functions that are turned off by default. This set of elementary functions only contains well-known functions that are frequently used by engineers. Note that most of the conventional GP implementations work with a much smaller set of elementary functions.

GGP test cases

Typical test cases for standard GP codes are functions like x-2x^3 that can easily be constructed by the three arithmetic operations +,-,* only. Note that the operation / is often turned off, because it can cause problems. GGP easily finds such functions with the default function set, although this set is much larger and contains / as well as the even more problematic functions pow and lga. Since GGP often finds the correct answer already in the first generation (containing 100 individuals by default), i.e., 'by chance', such simple functions are useless for testing GGP.

It has already been mentioned that functions like sin(3.14*x)/x are already hard for standard GP. The function sin(πx)/(πx) is a very nice function with zeros in the points x_i=i, except x_0=0 where it is equal to 1. If we compute sin(πx)/(πx) in 50 points in the interval x=0…1, it is hard to see from these data that sin(πx)/(πx) is the correct solution. Since GGP takes more time to evaluate an individual with its parameter optimization, comparing the number of individuals used by GP and GGP for solving a problem, would be unfair. Therefore, we consider the 'number of fitness computations' in the following. To evaluate the fitness of an individual, GGP performs n iterations in the parameter optimization loop. For each iteration, a fitness computation is made. Thus, the number of fitness computations is proportional to the program execution time when the fitness computation is time-consuming. However, traditional GP codes (with a population size of 500 individuals and a huge tree depth) were not able to solve the sin(πx)/(πx) test case within 50'000 fitness computations. Although some approximations were found, none of them was really close to the correct solution, even when the definition interval was extended to x=0…2. From the analysis of the results, one finds that there is almost no chance that GP could find the correct solution even with considerably more fitness computations. Like GP, GGP has some dependence on the initial random population, especially if a small population size is used. Therefore, several runs with different initial populations are required for obtaining some statistical description of the performance. Moreover, the success of GGP depends on the tree depth. For the test example, depth 2 would be sufficient, but with this depth GGP never found the correct answer. Already with depth 3, GGP typically found the correct answer with 1'000-5'000 fitness computations. Some typical GGP solutions of sin(3.1416x)/(3.1416x) are:

$$f_1^{GP} = 7.9513\text{E}3 * \sin(\sin(\text{e} * 1.4727\text{E}-5) * \sin(\pi) / \text{x})$$

$$f_2^{GP} = -87.339 * \sin(\sin(2 * 1.5702) * \sin(\pi) / (\sin(1.5702 * \text{e}) * \text{x} / \text{e}))$$

$$f_3^{GP} = 1.0004 * \text{sim}(3.1389 * \text{x}) / (\pi)$$

Note that it is not always easy to check that GGP has found a correct solution. GGP could also find correct answers with tree depth 4 and with variable tree depth 2-5. In both cases, more fitness computations were required than for depth 3. Figure 10 and Figure 11 illustrate the GGP search.

Another interesting test case is sin(c/x), where c is a constant, for example, 0.4. Again, this case is difficult because of the divide by zero for x=0. Moreover, the function oscillates rapidly close to x=0. If the number of observation points is finite, the sampling theorem must be violated near x=0. Thus, the classical Fourier series approach would fail. Far away from x=0, sin(c/x) is almost c/x, i.e., it becomes very smooth and does not oscillate at all. GGP could find correct answers when the definition interval was close to zero, but also when the definition interval was so far away that no oscillation was observed.

So far, we have simulated a set of observed data by a known mathematical function evaluated in a finite number of observation points. Consequently, the accuracy of the data was very high - much higher than in real life. What happens if the accuracy of the observations is reduced? We can simulate this by adding some random values to the given function values. This considerably disturbs not only genetic, but also conventional optimization procedures. If the number of observation points is small and if the observation interval is short, there can be many completely different solutions that fit within the given accuracy and neither GGP nor any other code can decide which of them is 'correct'.

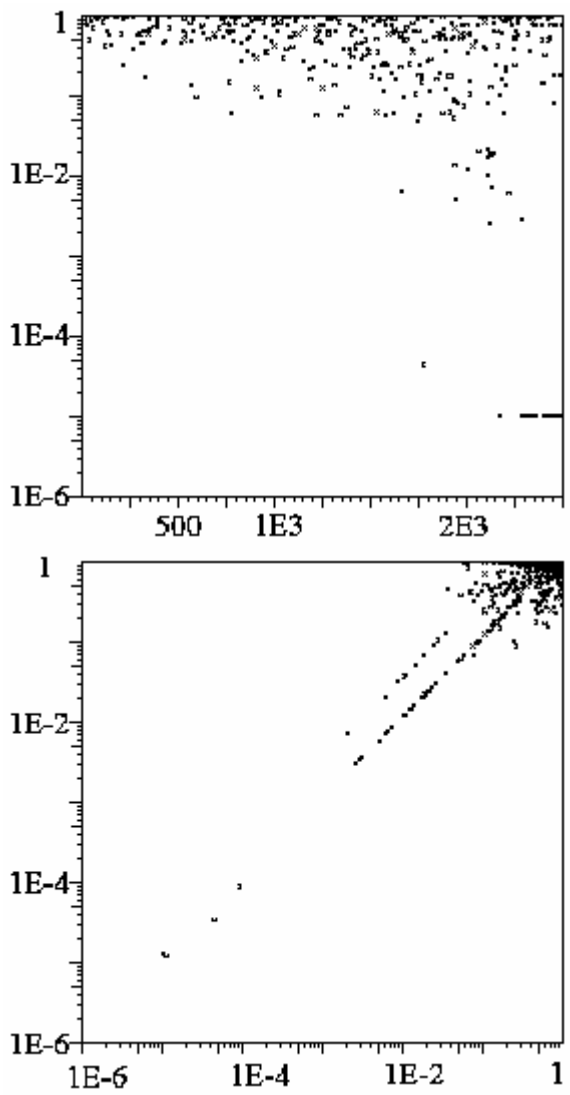

Figure 10: Typical GGP search run of the sin(3.1416x)/(3.1416x) test case with population size 100, formula tree depth 3, and GGP default settings. Left hand side: approximation error as a function of the fitness computation number. Right hand side: Extrapolation error versus approximation error. Here, the 'correct' solution has been found within 1'800 fitness computations. Note that the GGP search is concentrated along the diagonal, where one has 'balanced' individuals with an extrapolation of the same quality as the approximation.

However, for a sufficient number of observation points and a sufficiently large observation interval, GGP was still successful, although the performance slows down with decreasing accuracy of the observations.

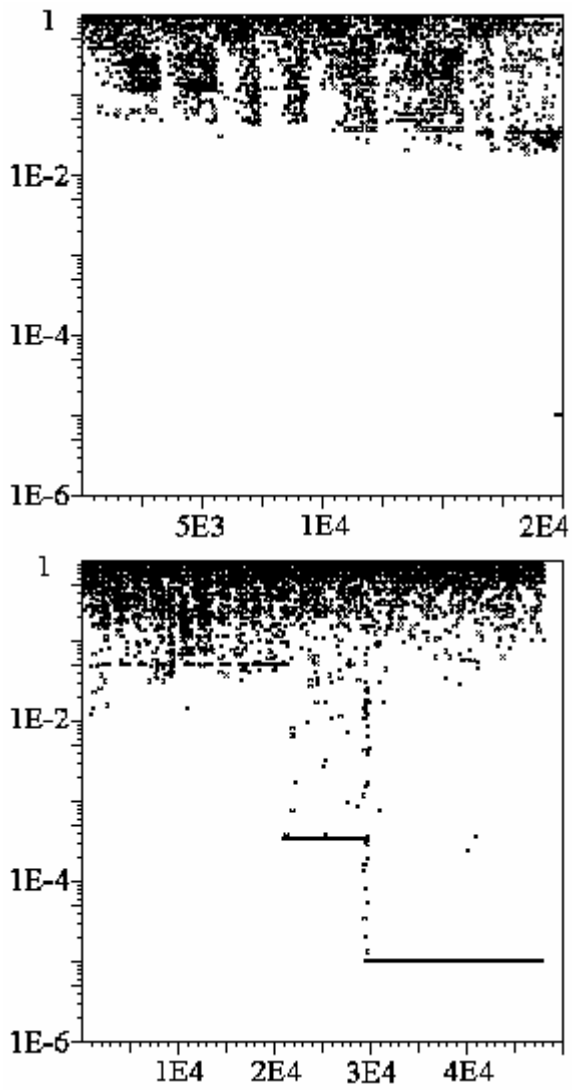

Figure 11: Same as Figure 10 for variable function tree depth 2-5 (left hand side) and for function tree depth 4. Here, the 'correct' solution has been found within 1'300 fitness computations. Note that much more iterations are required here. In both cases about 20'000 fitness computations were required for finding the correct solution. A stair-case behavior can be observed on the right hand side.

Although these results are very encouraging, there is still a difficult problem for GP and GGP with only one basis function. What happens when we observe, for example, the superposition of several independent resonators? In such situations, a generalized series expansion approach is promising. To tackle such problems with GGP, a combination with linear parameter optimization is required.

Further generalizations

Although one can consider GGP as a generalization of GP, its roots are completely different. GGP was obtained from a continuous generalization of Fourier series to generalized Fourier series, to generalized series expansions as outlined in the previous chapter. This generalization procedure was developed to illustrate the development of the MMP code for computational electromagnetics [Hafner, 1990/S], [Hafner, 1993/S]. Therefore, already the first version of GGP contained a generalized series expansion. In the following, the problems caused by this technique have to be solved. Note that generalization always cause problems. The success of the generalizations therefore depends on the solutions of these problems.

It has already been mentioned that the definition and computation of the fitness causes the main problem. If we consider each basis function as an individual, the series expansion is a group or a society. We can evaluate the **society fitness** exactly as we have computed the **individual fitness** before, i.e., from an analysis of the approximation and the extrapolation error. But now, we need a new definition of the individual fitness. First of all, this fitness is required for the genetic selection mechanisms acting on the individuals. Individuals who live in fit societies, obtain higher fitness than those who live in bad societies, but attaching the same fitness to all members of a society would certainly be wrong. How can we honor valuable members of a society? Naturally, the amplitude of a basis function gives some impression on the importance of the corresponding individual within a society, but this can be misleading. It has often been observed that two similar basis functions obtain big amplitudes of opposite sign. This corresponds to destructive work of the two individuals. Although the amount of work can be big, the resulting effect for the society can be small. Therefore, GGP **filter**s the **amplitude fitness** and it evaluates additional fitness values, named **error fitness** and **similarity fitness**. These fitness values check 1) how accurate the solution would be if the individual would do the job of the entire society and 2) how similar the individual is to other members of the society. Finally, the individual is computed as a **combined fitness** obtained from the society fitness, the amplitude fitness, the error fitness, and the similarity fitness.

What does a society do? First, it selects some individuals to become members. Of course, the society prefers individuals that are known to be fit. GGP has a **pool of individuals**, i.e., basis functions. The society can pick several individuals out of the pool. Initially, the fitness values of the individuals are unknown and the society is forced to randomly select individuals. Later on, the society prefers fitter individuals. GGP allows **cloning**, i.e., the society can contain several genetically identical individuals. Since the clones can have different parameter values, their phenotypes can be different. For GGP, traditional series expansions are societies consisting of clones of a few individuals only. For example, Fourier series work with clones of $\sin(p*x)$ and $\cos(p*x)$. Therefore, GGP is a generalization of GP as well as of Fourier and other series expansions.

Societies that strictly use the fittest individuals cause problems, because they generate many outsiders that never get a job. Although such societies can be efficient at the beginning, they turn out to be inefficient after a while. It makes no sense to generate an individual without using it. Therefore, GGP prohibits selecting used individuals as long as there are unused individuals in the pool.

From time to time, the worst individuals in the pool are killed and replaced by new ones like in the traditional GP. GGP does not only keep the fittest individuals like the well-known **elitist** strategy, it also saves the fittest individuals in **data files**. The society cannot only select individuals from the pool, it can also select them from data files. This is advantageous if one runs GGP several times on similar problems. In these cases, GGP can take advantage of the knowledge obtained in previous runs. Data files play a role similar to libraries. With data files,

GGP can clone individuals who were prominent a long time ago. I.e., GGP could generate a society consisting of clones of Einstein and Newton and it could generate children of them. Although we do not have this opportunity in real life, such features can be useful and efficient for solving problems on computers.

The individual-society structure and the various generalizations outlined in this section, make the program structure of GGP much more complex than the structure of traditional GP codes. The GGP structure is even so complex that it is hard to draw a flow diagram. Moreover, there are many system parameters and symbolic system formula that can be modified by the user for improving the GGP performance. Of course, this makes the description and handling of GGP difficult. Fortunately, default settings have found an implemented that were useful for many different test cases. Therefore, the user can simply run GGP as it is. No doubt, the default settings might be improved and adapted to specific situations.

The symbolic definition of fitness values, fitness filters, parameter optimization procedures, etc. allow a symbolic optimization of important parts of GGP by another, external GGP code. Such a self-optimization is attractive and interesting, but Pentium based PCs are too slow for doing that.

Further GGP tests

When one runs GGP with societies with more than one individual on the test cases presented above, GGP can optimize several individuals at the same time. This is especially interesting if the society fitness evaluation is much more time-consuming than the fitness evaluations of the members of the society. With an increasing society size, GGP works more and more like conventional series expansions. I.e., excellent approximations can be found very quickly (see Figure 12). As soon as the society size exceeds some value, the extrapolations become worse and it becomes harder to find 'correct' solutions. For the test cases discussed before, single individual societies, i.e., society size 1, is sufficient.

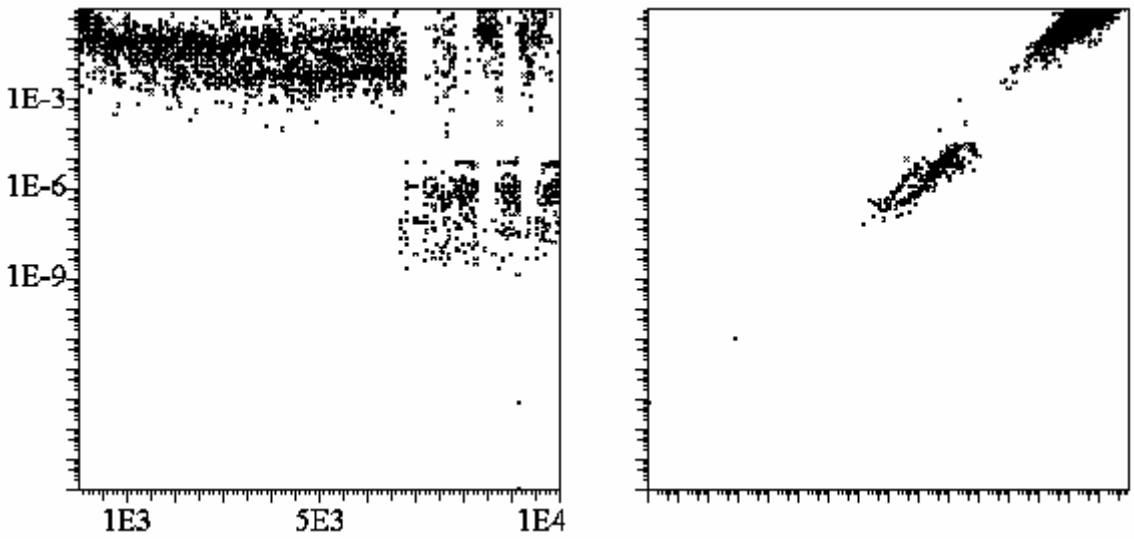

Figure 12: Same as Figure 10 for a GGP run with a society containing 3 individuals. Left hand side: approximation error versus fitness computation number. Note that an extremely accurate solution with an error of about 1E-16 has been obtained after 9'000 fitness computations, i.e., machine precision has been reached here. The figure on the right hand side shows again the extrapolation error versus the approximation error. The range of both axes is extremely wide, i.e., 1E-16…1. The search is still close to the diagonal, but almost all individuals are

considerably above the diagonal. The extrapolation error is typically 10…100 times bigger than the approximation error. I.e., on can already observe the trend toward accurate approximations and inaccurate extrapolations that is typical for series expansions and societies with many individuals.

Obviously, multi-member societies are much better suited when the observed function has the shape of a series expansion with a finite number of terms. If we receive a signal caused by several independent sources, we have a typical example. Human ear and brain have much experience with this situation and can easily separate the different sources, even if the corresponding amplitudes are very different. GGP can solve such problems when the amplitudes of the different sources are of similar magnitude and when the number of terms is relatively small. However, this class of problems is certainly quite demanding for our poor computers. If one does not know in advance, how many sources there are, it is better to start with a small society size and increasing the society size later on when GGP is not as successful as expected.

What happens, when the observed function cannot be constructed by an appropriate combination of the elementary functions implemented in GGP? To test GGP, a data file containing the data of a 'tricky' function was passed to GGP. With a society size of only 5, GGP found an excellent approximation with a quite good extrapolation. Surprisingly, all members of the best society were very similar, not clones, but 'cousins'. With some analytical simplification, it was found that GGP had constructed a series expansion of the type

$$f^0(x) = \sum_{k=1}^{K} (x / a_k)^{b_k / x} + error(x) \ , \tag{1.3}$$

i.e., the 'cousins' could be replaced by pure clones. In fact, the correct answer would have been erfc(2/3x), where erfc denotes the complementary error function. GGP found a strange series expansion that seems to be very good for x>0. Since the computation of the powers (contained in GGP's basis) is time-consuming on usual processors, such series expansions are numerically less efficient than continuous fractions. Nevertheless, this is a big success for GGP, because series expansions of the form (1.3) could not be found in mathematical literature, i.e., GGP has the potential of finding new types of series expansions.