10

Genetic Programming in Application to Flight Control System Design Optimisation

Anna Bourmistrova and Sergey Khantsis
Royal Melbourne Institute of Technology
Australia

1. Introduction

1.1 Evolutionary algorithms

EAs are often considered as an example of *artificial intelligence* and a *soft computing* approach. Their unique ability to search for complete and global solutions to a given problem makes EAs a powerful problem solving tool which combine such important characteristics as robustness, versatility and simplicity.

Historically, there exist several branches of EAs, namely Genetic Algorithms, Genetic Programming, Evolutionary Programming and Evolutionary Strategies. Their development started independently in the 1960s and 70s. Nevertheless, all of them are based on the same fundamental principle - evolution. 'Evolution' is used here in its Darwinian sense, the advance through 'survival of the fittest'.

Despite of a remarkable simplicity, EAs have proven to be capable of solving many practical tasks. The first and obvious application is numerical optimisation (minimisation or maximisation) of a given function. However, EAs are capable of much more than function optimisation or estimation of a series of unknown parameters within a given model of a physical system. Due to, in a large part, their stochastic nature, EAs can *create* such complex structures as computer programs, architectural designs and neural networks. Several applications of EAs have been known to produce a patentable invention (Koza et al., 1996, Koza et al., 1999 and Koza et al., 2003).

Unfortunately, such a truly intelligent application of EAs is rarely used for practical purposes. GP and similar algorithms often require a supercomputing power to produce an optimal solution for a practical task. This may be overcome, at least partially, by narrowing the search space.

A general engineering design practice is to propose a new design based on existing knowledge of various techniques (not uncommonly even from other fields) and no less important, intuition. Following this, the proposal is analysed, tried on a real system or its mathematical model, findings and errors are analysed again, the design is modified (or rejected) and the process continues until a satisfactory solution is found.

EAs work basically on the same principle, although, obviously, using less analytical analysis but more trial-and-error approach. It was found, however, that the process of selecting the most suitable solutions at each stage and producing the next iteration variants is, overall, largely intelligent and heuristic. EAs are capable to answer not only the question *how to do* something (how to control, in particular), but also the question *what to do* in order to meet

Source: New Achievements in Evolutionary Computation, Book edited by: Peter Korosec,
ISBN 978-953-307-053-7, pp. 318, February 2010, INTECH, Croatia, downloaded from SCIYO.COM

the objective. It is therefore appealing to apply such a promising automated technique to a problem with no general solution at hand.
The guidance and flight control is not a totally unstudied area where no convincing guesses can be made and where no parallels with the existing solutions are possible. This fact allows to watch, understand and guide, to a certain extent, the process of evolution. It also enables to optimise the EA for the purposes of control design.
The latter is especially useful because there are still very little research done on artificial evolution of structures of controllers in particular. An overwhelming majority of EA applications is concerned with numeric optimisation. A few proponents of a more advanced use (e.g. Koza et al., 2000, De Jong & Spears, 1993) are keen to show the real scope of possible applications, including controller design.

1.2 Unmanned aerial vehicles and shipboard recovery

Over the past two decades, the use of UAVs is becoming a well accepted technique not only for the military applications but also in the civilian arena. Typical applications of UAVs range from such traditional military missions as battlefield surveillance, reconnaissance and target acquisition to atmospheric research, weather observation, coastal and maritime surveillance, agricultural and geological surveying, telecommunication signals retranslation, and search and rescue missions.
The critical parts of a UAV mission are the launch and recovery phases. Although some UAVs can be conventionally operated from runways, the ability of UAVs to be operated from confined areas, such as remote land sites, ships and oil rigs, greatly increase their practical applications. Such operations generally require the aircraft to either have Vertical Take- Off and Landing (VTOL) capability or some form of launch and recovery assistance.
Unlike launch, the ways of UAV recovery are numerous. Probably the most widely used method, apart from runway landing, is the parachute assisted recovery. Unfortunately, parachute recovery can hardly be used when the landing area is extremely limited (for example, a ship's deck) and in the presence of high winds and strong gusts.
The first practicable and widely used solution for shipboard recovery of a fixed-wing UAV was capturing by an elastic net. This method has been employed for the USN *RQ-2 Pioneer* UAV, first deployed in 1986 aboard the battleship USS *Iowa*. The recovery net is usually stretched above the stern of the ship and the aircraft is flown directly into the net. A serious disadvantage of this method is that it is quite stressful for the aircraft. Nevertheless, due to simplicity of the recovery gear and reasonably simple guidance and flight control during the approach, this technique is still very popular for maritime operations.
Other methods include such techniques as deep stall and perched recovery and various forms of convertible airframes. However, these methods often imply very specific requirements to the UAV design and high complexity of control.
In this work a novel recovery method is proposed. This method, named *Cable Hook Recovery*, is intended to recover small to medium-size fixed-wing UAVs on frigate size vessels. It is expected to have greater operational capabilities than the *Recovery Net* technique, which is currently the most widely employed method of recovery for similar class of UAVs, potentially providing safe recovery even in very rough sea and allowing the choice of approach directions.
There are two distinct areas in recovery design: design of the recovery technique itself and development of a UAV controller that provides flight control and guidance of the vehicle in

accordance with the requirements of this technique. The controller should provide autonomous guidance and control during the whole recovery process (or its airborne stage).
It should be noted that there exists a number of control design techniques applicable to the area of guidance and flight control. They all have different features and limitations, producing the controllers with different characteristics. It is expected that linear control techniques will not be sufficient for control of the aircraft through the whole recovery stage due to large atmospheric disturbances, ship motion and aircraft constraints.
Under these conditions when so many factors remain uncertain during the process of development, even the very approach to the control problem is unclear. It is desirable that the controller design methodology allow to produce an optimally suitable controller even when faced with such uncertainties and that could be done with application of EAs.

2. Evolutionary algorithms

Over the hundred years of aviation history, various linear control methods have been successfully used in the aerospace area due to their simplicity and analytical justifiability. Despite their natural limitations, linear control techniques still remain as one of the most accepted design practices. However, growing demands to the performance of aircraft, and on the other hand, a remarkable increase in available computation power over the past years have led to significant growth if the popularity of nonlinear control techniques.
A principal difficulty of many nonlinear control techniques, which potentially could deliver better performance, is the impossibility or extreme difficulty to predict theoretically the behaviour of a system under all possible circumstances. Therefore, it becomes a challenging task to verify and validate the designed controller under all real flight conditions. There is a need to develop a consistent nonlinear control design methodology that enables to produce a required controller for an arbitrary nonlinear system while assuring its robustness and performance across the whole operational envelope at the same time.
The Evolutionary Algorithms (EAs) is a group of such stochastic methods which combine such important characteristics as robustness, versatility and simplicity and, indeed, proved the success in many applications, such as neural network optimisation (McLean &. Matsuda, 1998), finance and time series analysis (Mahfoud & Mani, 1996), aerodynamics and aerodynamic shape optimisation (McFarland & Duisenberg, 1995), automatic evolution of computer software and, of course, control (Chipperfield & Flemming, 1996).

2.1 Introduction to evolutionary algorithms

Evolutionary algorithm is an umbrella term used to describe computer-based problem solving systems which employ computational models of evolutionary processes as the key elements in their design and implementation. All major elements found in natural evolution are present in EAs. They are:

- Population, which is a set of individuals (or members) being evolved;
- Genome, which is all the information about an individual encoded in some way;
- Environment, which is a set of problem-specific criteria according to which the fitness of each individual is judged;
- Selection, which is a process of selecting of the fittest individuals from the current population in the environment;
- Reproduction, which is a method of producing the offspring from the selected individuals;

- Genetic operators, such as mutation and recombination (crossover), which provide and control variability of the population.

The process of evolution takes a significant number of steps, or generations, until a desired level of fitness is reached. The 'outcome' should not be interpreted as if some particular species are expected to evolve. The evolution is not a purposive or directed process. It is expected that highly fit individuals will arise, however the concrete form of these individuals may be very different and even surprising in many real engineering tasks. None of the size, shape, complexity and other aspects of the solution are required to be specified in advance, and this is in fact one of the great advantages of the evolutionary approach. The problem of initial guess value rarely exists in EA applications, and the initial population is sampled at random.

The first dissertation to apply genetic algorithms to a pure problem of mathematical optimisation was Hollstien's work (Hollstien, 1971). However, it was not until 1975, when John Holland in his pioneering book (Holland, 1975) established a general framework for application of evolutionary approach to artificial systems, that practical EAs gained wide popularity. Until present time this work remains as the foundation of genetic algorithms and EAs in general.

Now let us consider the very basics of EAs. A typical pseudo code of an EA is as follows:

```
Create a {usually random} population of individuals;
Evaluate fitness of each individual of the population;
until not done {certain fitness, number of generations etc.}, do
Select the fittest individuals as 'parents' for new generation;
Recombine the 'genes' of the selected parents;
Mutate the mated population;
Evaluate fitness of the new population;
end loop.
```

It may be surprising how such a simple algorithm can produce a practical solution in many different applications.

Some operations in EAs can be either stochastic or deterministic; for example, selection may simply take the best half of the current population for reproduction, or select the individuals at random with some bias to the fittest members. Although the latter variant can sometimes select the individuals with very poorly fitness and thus may even lead to temporary deterioration of the population's fitness, it often gives better overall results.

2.2 Evolutionary algorithm inside

There are several branches of EAs which focus on different aspects of evolution and have slightly different approaches to ongoing parameters control and genome representation.

In this work, a mix of different evolutionary methods is used, combining their advantages. These methods have the common names: Genetic Algorithms (GA), Evolutionary Programming (EP), Evolutionary Strategies (ES) and Genetic Programming (GP).

As noted above, all the evolutionary methods share many properties and methodological approaches.

2.2.1 Fitness evaluation

Fitness, or objective value, of a population member is a degree of 'goodness' of that member in the problem space. As such, fitness evaluation is highly problem dependent. It is implemented in a function called fitness function, or more traditionally for optimisation

methods, objective function. The plot of fitness function in the problem space is known as fitness landscape.
The word 'fitness' implies that greater values ('higher fitness') represent better solutions. However, mathematically this does not need to be so, and by optimisation both maximisation and minimisation are understood.
Although EAs are proved themselves as robust methods, their performance depends on the shape of the fitness landscape. If possible, fitness function should be designed so that it exhibits a gradual increase towards the maximum value (or decrease towards the minimum). In the case of GAs, this allows the algorithm to make use of highly fit building blocks, and in the case of ESs – to develop an effective search strategy quickly.
In solving real world problems, the fitness function may be affected by noise that comes from disturbances of a different nature. Moreover, it may be unsteady, that is, changing over time. Generally, EAs can cope very well with such types of problem, although their performance may be affected.
It is accepted that the performance of an optimisation algorithm is measured in terms of objective function evaluations, because in most practical tasks objective evaluation takes considerably more computational resources than the algorithm framework itself. For example, in optimisation of the aerodynamic shape of a body, each fitness evaluation may involve a computer fluid flow simulation which can last for hours. Nevertheless, even for simple mathematical objective functions EAs are always computationally intensive (which may be considered as the price for robustness).
The computation performance may be greatly improved by parallelisation of the fitness evaluation. EAs process multiple solutions in parallel, therefore they are extremely easily adopted to parallel computing.
Another useful consequence of maintaining a population of solutions is the natural ability of EAs to handle multi-objective problems. A common approach – used in this work - to reduce a multiobjective tasks to a single-objective one, by summing up all the criteria with appropriate weighting coefficients, is not always possible. Unlike single point optimisation techniques, EAs can evolve a set of Pareto optimal solutions simultaneously. A Pareto optimal solution is the solution that cannot be improved by any criterion without impairing it by at least one other criterion, which is a very interesting problem on its own.

2.2.2 Genome representation

In EA theory, much as well as in natural genetics, genome is the entire set of specifically encoded information that fully defines a particular individual. This section focuses only on numeric genome representation. However, EAs are not limited to numeric optimisations, and for more 'creative' design tasks a more sophisticated, possibly hierarchical, genome encoding is often required. In fact, virtually any imaginable data structure may be used as a genome. This type of problem is addressed in Section 2.5 Genetic Programming.
A commonly used genome representation in GAs is a fixed length binary string, called chromosome, with real numbers mapped into integers due to convenience of applying genetic operators, recombination (crossover) and mutation.
Accuracy is a common drawback of digital (discrete) machines, and care should be taken when choosing appropriate representation. For example, if *8* bit numbers are used and the problem determines the range of *[–1.0; 1.0]*, this range will be mapped into integers *[00000000; 11111111] ([0; 255]* decimal)1. As *255* corresponds to *1.0*, the next possible integer, *254*, will correspond to *0.99215686*, and there is no means of specifying any number between *0.99215686* and *1.0*.

The required accuracy is often set relative to the value, not the range. In this case, linear mapping may give too low accuracy near zero values and too high accuracy towards the end of the range. This was the reason for inventing the so called floating point number representation, which encodes the number in two parts: mantissa, which has a fixed range, and an integer exponent, which contain the order of the number. This representation is implemented in nearly all software and many hardware platforms which work with real numbers.

Another type of genome encoding which should be mentioned is the permutation encoding. It is used mostly in ordering problems, such as the classic Travelling Salesman Problem, where the optimal order of the given tokens is sought after. In this case, a chromosome can represent one of the possible permutations of the tokens (or rather their codes), for example, '*0123456789*' or '*2687493105*' for ten items. When this type of encoding is used, special care should be taken when implementing genetic operators, because 'traditional' crossover and mutation will produce mostly incorrect solutions (with some items included twice and some items missing).

Finally, a vector of floating point real values can be used as a chromosome to represent the problem that deals with real values. However, domain constraints handling should be implemented in most cases, as the floating point numbers, unlike integers, have virtually no highest and lowest values (in a practical sense). Moreover, special recombination and mutation operations should be used in this case.

2.2.3 Selection

Selection is the key element of all evolutionary algorithms. During selection, a generally fittest part of the population is chosen and this part (referred as mating pool) is then used to produce the offspring.

The requirements for the selection process are somewhat controversial. On the one hand, selection should choose 'the best of the best' to increase convergence speed. On the other hand, there should be some level of diversity in the population in order to allow the population to develop and to avoid premature convergence to a local optimum. This means that even not very well performing individuals should be included in the mating pool.

This question is known as the conflict between exploitation and exploration. With very little genetic diversity in a population, new areas in the search space become unreachable and the process stagnates. Although exploration takes valuable computation resources and may give negative results (in terms of locating other optima), it is the only way of gaining some confidence that the global optimum is found (Holland, 1975).

It should be noted that the optimal balance between exploitation and exploration is problem dependent. For example, real-time systems may want a quick convergence to an acceptable sub-optimal solution, thus employing strong exploitation; while engineering design which uses EAs as a tool is often interested in locating various solutions across the search space, or may want to locate exactly the global optimum. For the latter tasks, greater exploration and thus slower convergence is preferred.

Balance between exploitation and exploration can be controlled in different ways. For example, intuitively, stronger mutation favours greater exploration. However, it is selection that controls the balance directly. This is done by managing the 'strength' of selection. Very strong selection realises exploitation strategy and thus fast convergence, while weak selection allows better exploration.

A general characteristic that describes the balance between 'perfectionism' and 'carelessness' of selection is known as selection pressure or the degree to which the better individuals are favoured. Selection pressure control in a GA can be implemented in different ways; a very demonstrative parameter is the size of the mating pool relative to the size of the population. As a rule, the smaller the mating pool, the higher the selection pressure.
A quantitative estimation of the selection pressure may be given by the take-over time (Goldberg & Deb, 1991). With no mutation and recombination, this is essentially the number of generations taken for the best member in the initial generation to completely dominate the population.
The efficiency of one or another selection method used in EAs largely depends on population properties and characteristics of the whole algorithm. A theoretical comparison of the selection schemes may be found in (Goldberg & Deb, 1991).
First, all selection methods can be divided into two groups: stochastic and deterministic. Deterministic methods use the fitness value of a member directly for selection. For example, the best half of the population may be selected or all individuals with the fitness better than a given value may be included in the mating pool. In contrast, stochastic selection methods use fitness only as a guide, giving the members with better fitness more chances to be selected allowing greater exploration. However, deterministic methods can also be tuned to allow greater exploration. For example, every second member in a sorted by fitness list can be taken for reproduction instead of the best half of the population.
One of the simple ways to reduce the possible impact of stochastic sampling errors is to guarantee that the best, or elite, member(s) is always selected for reproduction. This approach is known as Elitism. In effect, elitism is the introduction of a portion of deterministic selection into a stochastic selection procedure. In most cases, elitism assumes that the elite member is not only selected, but also propagated directly to the new population without being disrupted by recombination or mutation. This approach ensures that the best-so-far achievement is preserved and the evolution does not deteriorate, even temporarily.
Another feature which may be applied to any selection scheme is population overlapping. The fraction of the old population which is replaced with the fresh members is called a generation gap (De Jong, 1975). Nothing particularly advantageous is found in overlapping schemes, although they may be useful for some problems, in particular for steady and noiseless environments (Goldberg & Deb, 1991).
It should be also noted that some reproduction schemes allow multiple selection of one member, while others do not. The former case (also referred to as replacement) means that the selected member is returned back to the mating pool and thus may be selected again in the same generation. This feature is often used in stochastic selection methods such as fitness proportional selection and tournament selection.

2.2.3.1 *Deterministic selection*

All members are straightforwardly sorted according to their fitness value and some of the best members are picked up for reproduction. A certain number of members (or population percentage) is usually chosen, or the members may be selected one by one and reproduced until the next generation population is filled up.
As noted before, deterministic methods are more suitable for the algorithms with small populations (less than about *20–40* members). They are therefore used in the areas where small populations are desirable (e.g. when fitness evaluation is extremely costly) or where it is traditionally adopted (in particular in Evolutionary Strategies, see Section 2.4).

2.2.3.2 *Fitness proportional selection and fitness scaling*

In fitness proportional selection, all individuals receive the chances to reproduce that are proportional to their objective value (fitness). There are several implementations of this general scheme which vary in stochastic properties and time complexity: roulette wheel (Monte Carlo) selection, stochastic remainder selection and stochastic universal selection. The roulette wheel method is described here in more detail.

The analogy with a roulette wheel arises because one can imagine the whole population forming a roulette wheel with the size of any individual's slot proportional to its fitness. The wheel is then spun and the 'ball' thrown in. The probability of the 'ball' coming to rest in any particular slot is proportional to the arc of the slot and thus to the fitness of the corresponding individual (Coley, 1999).

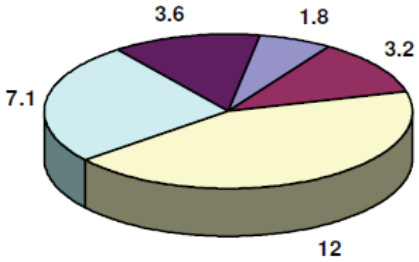


Fig. 1. Roulette wheel selection

There are no means to control the selection pressure and the convergence speed: they are determined entirely by the fitness of each individual.

However, such a control is often necessary. If for example, a fit individual is produced, fitness proportional selection with replacement can allow a large number of copies of this individual to flood the subsequent generations.

One of the methods intended to overcome this problem and to maintain a steady selection pressure is linear fitness scaling (Coley, 1999). Linear fitness scaling works by pivoting the fitness of each individual about the average population fitness. The scale is chosen so that an approximately constant proportion of copies of the best members is selected compared to the 'average member'.

There are some more sophisticated scaling techniques, such as sigma scaling (Coley, 1999), in which the (expected) number of trials each member receives is adjusted according to the standard deviation of the population fitness.

2.2.3.3 *Ranking selection*

This is a development of the fitness proportional selection, aimed to achieve greater adaptability and to reduce stochastic. The idea represents the combination of fitness proportional and deterministic selection. The population is sorted according to the fitness, and a rank is assigned to each individual. After assigning the rank, a proportionate selection is applied as described in the previous section, using rank values instead of fitness.

Ranking has two main advantages before fitness proportional selection (even that with fitness scaling). First, the required selection pressure can be controlled more flexibly by applying a specific rank assignment function. Second, it softens stochastic errors of the search, which can be especially destructive for the fitness functions affected by noise. If a

particularly fit member is generated that stands well off the whole population. Even if the proportionate selection is constrained by fitness scaling, this best member will be greatly favoured, whilst the rest of the population will receive very low selection pressure because the differences between their fitness values are insignificant as compared to the 'outlier'. In contrast, ranking will establish a predefined difference between the neighbouring members, ensuring an adequate selection pressure for the whole population.

2.2.3.4 *Tournament selection*

Tournament selection is a simple yet flexible stochastic selection scheme. Choose some number s of individuals randomly from a population and then select the best individual from this group. Repeat as many times as necessary to fill up the mating pool. This somewhat resembles the tournaments held between s competitors. As a result, the mating pool, being comprised of tournament winners, has a higher average fitness than the average population fitness.
The selection pressure can be controlled simply by choosing appropriate tournament size s. Obviously, the winner from a larger tournament will, on average, have a higher fitness than the winner of a smaller tournament. In addition, selection pressure can be further adjusted via randomisation of the tournament.

2.2.4 Recombination

Recombination allows solutions to exchange the information in a way similar to that used by a biological organism undergoing sexual reproduction. This effective mechanism allows to combine parts of the solution (building blocks) successfully found by parents. Combined with selection, this scheme produces, on average, fitter offspring. Of course, being a stochastic operation, recombination can produce 'disadvantaged' individuals as well; however, they will be quickly perished under selection.
Recombination is usually applied probabilistically with a certain probability. For GAs, the typical value is between *0.6* and *0.8*; however, the values up to *1.0* are common.

2.2.4.1 *Alphabet based chromosome recombination*

In essence, recombination is 'blending' the information of two (or more) genomes in some way. For typical GAs, an approach from natural genetics is borrowed. It is known as crossover. During crossover, chromosomes exchange equal parts of themselves. In its simplest form known as single-point crossover, two parents are taken from the mating pool. A random position on the chromosome (locus) is chosen. Then, the end pieces of the chromosomes, starting from the chosen locus, are swapped.
Single-point crossover can be generalised to k-point crossover, when k different loci are chosen and then every second piece is swapped. However, according to De Jong's studies (De Jong, 1975) and also (Spears & Anand, 1991), multi-point crossover degrades overall algorithm performance increasingly with an increased number of cross points.
There is another way of swapping pieces of chromosomes, known as uniform crossover. This method does not select crossover points. Instead, it considers each bit position of the two parents one by one and swaps the corresponding bits with a probability of *50%*. Although the uniform crossover is, in a sense, an extreme case of multi-point crossover and can be considered as the most disruptive its variant, both theoretical and practical results (Spears & Anand, 1991) show that uniform crossover outperforms k-point crossover in most cases.

2.2.4.2 *Real value recombination*

Real-coded EAs require a special recombination operator. Unlike bit strings, real parameters are not deemed as strings that can be cut into pieces. Instead, they are processed as a whole in a common mathematical way. Due to rather historical reasons, real-coded EAs were mostly developing under the influence of Evolutionary Strategies and Evolutionary Programming (see Section 2.4). As a result, real-value recombination has not been properly considered until the fairly recent past ('90s). Nevertheless, a number of various recombination techniques have been developed. Detailed analysis of them is available in (Beyer & Deb, 2001, Deb et al., 2001 and Herrera et al., 1998).

The simplest real-value recombination one can think of is the averaging of several parents, which is known as arithmetic crossover. This method produces one offspring from two or more parents. Averaging may be weighted according to parents' fitness or using random weighting coefficients.

Self-adaptive recombination create offspring statistically located in proportion to the difference of the parents in the search space. These recombination operators generate one or two children according to a probability distribution over two or more parent solutions where if the difference between the parent solutions is small, the difference between the child and parent solutions should also be small.

The most popular approach is to use a uniform probability distribution—the so called 'blend crossover', *BLX*. The *BLX* operator randomly picks a solution in the range

$$\left[x_1 - \alpha\left(x_2 - x_1\right); x_2 + \alpha\left(x_2 - x_1\right)\right] \tag{1}$$

for two parents $x_1 < x_2$. α is the parameter which controls the spread of the offspring interval beyond the range of the parents' interval $[x_1; x_2]$.

Other approaches suggest non-uniform probability distribution. The Simulated Binary Crossover (*SBX*) uses a bimodal probability distribution with its mode at the parent solutions. It produces two children from two parents. This technique has been developed by K. Deb and his students in 1995. As the name suggests, the *SBX* operator simulates the working principle of the single-point crossover operator on binary strings.

A different approach is demonstrated by the Unimodal Normal Distribution Crossover (*UNDX*) (Ono & Kobayashi, 1997). It uses multiple (usually three) parents and create offspring solutions around the centre of mass of these parents. *UNDX* uses a normal probability distribution, thus assigning a small probability to solutions away from the centre of mass. Another mean-centric recombination operator is the Simplex Crossover (*SPX*). It differs from *UNDX* by assigning a uniform probability distribution for creating offspring in a restricted region marked by the parents.

Although both mean-centric and parent-centric recombination methods were found to exhibit self-adaptive behaviour for real-coded GAs similar to that of ESs (see Section 2.4), in a number of reports parent-centric methods were found generally superior (Deb et al., 2001).

2.2.5 Mutation

Mutation is another genetic operator borrowed from nature. However, unlike recombination, which is aimed at producing better offspring, mutation is used to maintain genetic diversity in the population from one generation to the next in a explorative way.

Not unlike recombination, mutation works differently for alphabet-based chromosomes and real-coded algorithms. However, in both cases it is merely a blind variation of a given individual.

2.2.5.1 *Bit string mutation*

In nearly all ordinary GAs, mutation is implemented as variation of a random bit in the chromosome. Each bit in the chromosome is considered one by one and changed with certain probability P_m. Bit change can be applied either as flipping (inverting) of the bit or replacing it with a random value. In the latter case, the actual mutation rate will be twice as low, because, on average, half of the bits are replaced with the same value.
In GAs, mutation is usually controlled through mutation probability P_m. As a rule, Gas put more stress on recombination rather than on mutation, therefore typical mutation rates are very low, of the order of *0.03* and less (per bit). Rates close to *0.001* are common. Nevertheless, mutation is very important because it prevents the loss of building blocks which cannot be recovered by recombination alone.
Mutation probability can be supressed in a number of ways. However, this may easily have an adverse effect—mutation is known for the ability to 'push' the stagnated process at the later stages.
Another technique to avoid stagnation is so called 'hypermutation'. Hypermutation is a method in which mutation probability is significantly (*10–100* times) increased for a small number of generations (usually one) during the run, or such a high rate is applied constantly to a certain part of the population (e.g. *20%*). Both hypermutation and random immigrants techniques are especially effective for dynamic environments, where the fitness landscape can change significantly over time (Grefenstette, 1999).
The above mutation control methods have a common drawback: they are not selective. Sometimes an individual approach may be desirable. In particular, stronger mutation might be applied to the weakest members, while less disruptive mutation to the best individuals. Alternatively, some more sophisticated methods can be developed. Such fine tuning is a more common feature of Evolutionary Strategies (see section 2.4) and real-coded GAs.
Some more direct control methods utilise additional bits in the genotype which do not affect the phenotype directly. However, these bits control the mutation itself. In the simplest case, a single flag bit can control the applicability of mutation to the respective parameter when zero value disables mutation and unity enables it.

2.2.5.2 *Real value mutation*

Implementation of a real-value mutation is rather more straightforward than that of alphabet strings. Unlike the latter, it is not an operation directly inspired by nature; however, as the real-coded algorithms generally do not use tricky encoding schemes and have the same problem-space and genotype values of the parameters, real-value mutation can be considered as the operation working on a higher level, up to direct phenotype mutation for function optimization problems.
Mutation to a real value is made simply by adding a random number to it. It is evident that the strength of real-value mutation can be controlled in a very convenient way, through the variance of the distribution (or the window size for the uniform distribution). Like with the common GAs that operate string-type chromosomes, mutation strength can be adapted using many different techniques, from simple predefined linear decrease to sophisticated adaptation strategies

2.2.6 Measuring performance

Performance of an EA is the measure how effective the algorithm is in search of the optimal solution. As evolutionary search is a stochastic and dynamic process, it can hardly be positively measured by a single figure. A better indicator of the performance is a convergence graph, that is, the graph 'fitness vs. computation steps'.

This figure presents two typical convergence graphs of two independent runs of the same GA with different sets of parameters. It can be seen that although the first run converges faster, it stagnates too early and does not deliver the optimal solution. Therefore, not the convergence speed nor time to reach a specified value, nor any other single parameter can be considered as a sole performance measure.

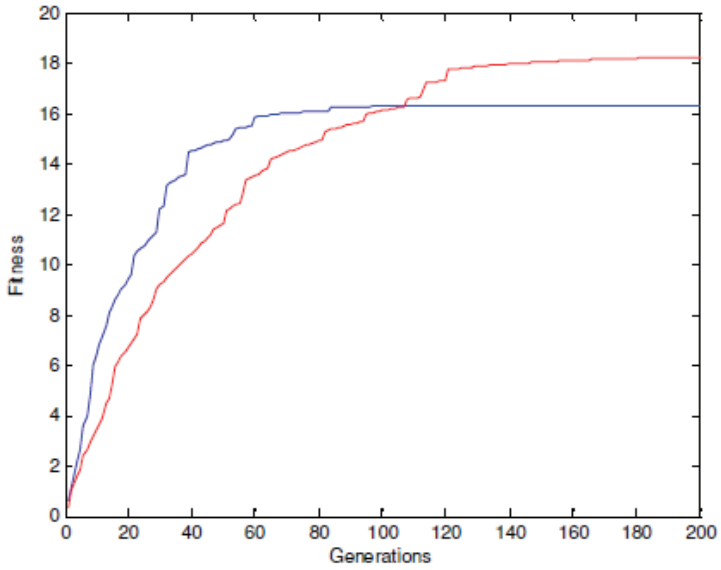


Fig. 2. Typical convergence graphs

De Jong in (De Jong, 1975) used two measures of the progress of the GA: the off-line performance and the on-line performance. The off-line performance is represented by the running average of the fitness of the best individual, f_{max}, in each population:

$$f_{off}(g) = \frac{1}{g}\sum_{i=1}^{g} f_{\max}(g) \tag{2}$$

where g is the generation number. In contrast, the on-line performance is the average of all fitness values calculated so far:

$$f_{on}(g) = \frac{1}{g}\sum_{i=1}^{g} f_{avg}(g) \tag{3}$$

The on-line performance includes both good and bad guesses and therefore reflects not only the progress towards the optimal solution, but also the resources taken for such progress. Another useful measure is the convergence velocity:

$$V(g) = \ln \sqrt{\frac{f_{\max}(g)}{f_{on}(1)}} \tag{4}$$

In the case of a non-stationary dynamic environment, the value of previously found solutions is irrelevant at the later steps. Hence, a better measure of optimisation in this case is the current-best metric instead of running averages.
When comparing the performance of different algorithms, it is better to use the number of fitness evaluations instead of the number of generations as the argument for performance characteristics.

2.2.7 The problem of convergence and genetic drift

Usually, the progress of EAs is fast at first and then loses its speed and finally stagnates. This is a common scenario for optimisation techniques. However, progressing too quickly due to greedy exploitation of good solutions may result in convergence to a local optimum an thus in low robustness. Several methods that help to control the convergence have been described in the above sections, including selection pressure control and adaptation of recombination and mutation rates.
However, it is unclear to which degree and how in particular to manage the algorithm's parameters. What is 'too fast' convergence? Unfortunately, the current state of EA theory is inadequate to answer this question *before* the EA is initiated.
For the most of the real-world problems, it takes several trial runs to obtain an adequate set of parameters. The picture of convergence process, as noted before, is not a good indicator of the algorithm's characteristics. In contrast, the plot of genetic diversity of the population against generation number (or fitness function evaluations) gives a picture which can explain some performance problems. If the population quickly loses the genetic diversity, this usually means a too high initial selection pressure. The further saturation may be attributed to reduction of selection pressure at the later stages. The loss of genetic diversity is known as genetic drift (Coley, 1998).

2.2.8 Schema theory

Schema theory has been developed by John Holland (Holland, 1975) and popularised by David Goldberg (Goldberg, 1989) to explain the power of binary-coded genetic algorithms. More recently, it has been extended to real-value genome representations (Eshelman & Schaffer, 1993) and tree-like *S*-expression representations used in genetic programming (Langdon & Poli, 2002, O'Reilly & Oppacher, 1995). Due to the importance of the theory for understanding GA internals, it is mentioned here, though it is not within the scope of this work to discuss it in details.

2.3 Genetic algorithms

Genetic algorithms are one of the first evolutionary methods successfully used to solve practical problems, and until now they remain one of the most widely used EAs in the engineering field. John Holland in (Holland, 1975) provided a general framework for GAs and a basic theoretical background, much of which has been discussed in the former sections. There are more recent publications on the basics of GA, for example (Goldberg, 1989).
The basic algorithm is exactly as in the Section 2.1; however, several variations to this scheme are known. For example, Koza (Koza, 1992) uses separate threads for asexual

reproduction, crossover and mutation, chosen at random; therefore, only one of these genetic operators is applied to an individual in each loop, while classical GA applies mutation after crossover independently.
One of the particularities of typical GAs is genome representation. The vast majority of GAs use alphabet-based string-like chromosomes described in Section 2.2.2, although real coded GAs are gaining wider popularity. Therefore, a suitable mapping from actual problem space parameters to such strings must be designed before a genetic search can be conducted.
The objective function can also have the deceptive properties as in most practical cases little is known about the fitness landscape. Nevertheless, if the fitness function is to be designed for a specific engineering task (for example, an estimate of the flight control quality, as will be used in this study later), attention should be paid to avoiding rugged and other GA-difficult fitness landscapes.
Of the two genetic operators, recombination (crossover) plays the most important role in Gas. Typical probability of crossover is *0.6* to *0.8* and even up to *1.0* in some applications. On the contrary, mutation is considered as an auxiliary operator, only to ensure that the variability of the population is preserved. Mutation probabilities range from about *0.001* to *0.03* per bit.
Population size is highly problem dependent; however, typical GAs deal with fairly large or at least moderate population sizes, of the order of *50* to *300* members, although smaller and much larger sizes (up to several thousands) could be used.
Although by far the largest application of GAs is optimisations of different sorts, from simple function optimisations to multi-parameter aerodynamic shape optimisation (McFarland & Duisenberg, 1995) and optimal control (Chipperfield & Flemming, 1996), GAs are suitable for many more tasks where great adaptation ability is required, for example, neural networks learning (Sendhoff & Kreuz, 1999) and time series prediction (Mahfoud & Mani, 1996). The potential of GA application is limited virtually only by the ability to develop a suitable encoding.

2.4 Evolutionary strategies and evolutionary programming

Evolutionary Programming was one of the very first evolutionary methods. It was introduced by Lawrence J. Fogel in the early 1960s (Fofel, 1962), and the publication (Fogel et al., 1966) by Fogel, Owens and Walsh became a landmark for EP applications. Originally, EP was offered as an attempt to create artificial intelligence. It was accepted that prediction is a keystone to intelligent behaviour, and in (Fogel et al., 1966) EP was used to evolve finite state automata that predict symbol strings generated from Markov processes and non-stationary time series.
In contrast, Evolutionary Strategies appeared on the scene in an attempt to solve a practical engineering task. In 1963, Ingo Rechenberg and Hans-Paul Schwefel were conducting a series of wind tunnel experiments in Technical University of Berlin trying to optimise aerodynamic shape of a body. This was a laborious intuitive task and the students tried to work strategically. However, simple gradient and coordinate strategies have proven to be unsuccessful, and Rechenberg suggested to try random changes in the parameters defining the shape, following the example of natural mutations and selection.
As it can be seen, both methods are focusing on behavioural linkage between parents and the offspring rather than seeking to emulate specific genetic operators as observed in nature. In addition, unlike GAs, natural real-value representation is predominantly used. In the

present state, EP and ES are very similar, despite their independent development over 30 years, and the historical associations to finite state machines or engineering field are no longer valid. In this study, ES approach is employed, so further in this section Evolutionary Strategies are described, with special notes when the EP practice is different.

2.4.1 Self-adaptation

One of the most important mechanisms that differs ES from the common GAs is endogenous control on genetic operators (primarily mutation). Mutation is usually performed on real-value parameters by adding zero mean normally distributed random values. The variance of these values is called step size in ES.

The adaptation of step size rules can be divided into two groups: pre-programmed rules and adaptive, or evolved, rules. The pre-programmed rules express a heuristic discovered through extensive experimentation. One of the earliest examples of pre-programmed adaptation is Rechenberg's (1973) *1/5* rule. The rule states that the ratio of successful mutations to all mutations should be *1/5* measured over a number of generations. The mutation variance (step size) should increase if the ratio is above *1/5*, decrease if it is below and remain constant otherwise. The variance is updated every k generations according to:

$$\sigma^{(g)} = \begin{cases} \sigma^{(g-k)} / c & n_s > 1/5 \\ \sigma^{(g-k)} \cdot c & n_s < 1/5 \\ \sigma^{(g-k)} & n_s = 1/5 \end{cases} \tag{5}$$

where n_s is the ratio of successful mutations and $0.817 \; c < 1$ is the adaptation rate. The lower bound $c = 0.817$ has been theoretically derived by Schwefel for the sphere problem (Ursem, 2003). The upper index in parenthesis henceforth denotes the generation number.

The other approach is the self-adaptive (evolved) control where Schwefel (Schwefel, 1981) proposed to incorporate the parameters that control mutation into the genome. This way, an individual $\boldsymbol{a} = (\boldsymbol{x}, \sigma)$ consists of *object variables* (sometimes referred as *describing parameters*) $\boldsymbol{x}$ and *strategy parameters* σ. The strategy parameters undergo basically the same evolution as object variables: they are mutated and then selected together, though only on the basis of objective performance, on which strategy parameters have indirect influence. The underlying hypothesis in this scheme is that good solutions carry good strategy parameters; hence, evolution discovers good parameters while solving the problem.

2.5 Genetic programming

Genetic programming (GP) is an evolutionary machine learning technique. It uses the same paradigm as genetic algorithms and is, in fact, a generalisation of GA approach. GP increases the complexity of the structures undergoing evolution. In GP, these structures represent hierarchical computer programs of varying size and shape.

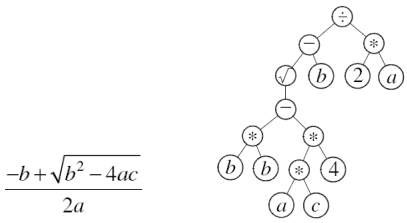
GP is a fairly recent EA method compared to other techniques discussed before in this chapter. The first experiments with GP were reported by Stephen Smith (Smith, 1980) and Nichael Cramer (Gramer, 1985). However, the first seminal book to introduce GP as a solid and practical technique is John Koza's 'Genetic Programming', dated 1992.

In GP, each individual in a population is a program which is executed in order to obtain its fitness. Thus, the situation is somewhat opposite to GAs: the individual is a 'black box' with

an arbitrary input and some output. The fitness value (often referred to as *fitness measure* in GP) is usually obtained through comparison of the program's output with the desired output for several input test values (*fitness cases*). However, fitness evaluation in GP is problem dependent and may be carried out in a number of ways. For example, the fitness of a program controlling a robotic animal may be calculated as the number of food pieces collected by the animal minus resources taken for search (e.g. path length). When seeking a function to fit the experimental data, the deviation will be the measure of fitness.
One of the characteristics of GP is enormous size of the search space. GP search in the space of possible computer programs, each of which is composed of varying number of functions and terminals. It can be seen that the search space is virtually incomprehensible, so that even generation of the initial random population may represent some difficulties. Due to that, GP typically works with very large populations of hundreds and even thousands of members. Two-parent crossover is usually employed as the main genetic operator, while mutation has only a marginal role or is not used at all.

2.5.1 Genome representation in GP and S-expressions

Unlike linear chromosomes in GAs, genomes in GP represent hierarchical, tree-like structures. Any computer program or mathematical expression can be depicted as a tree structure with functions as nodes and terminals as leaves. For example, let us consider an expression for one of the roots of a square equation $ax^2 + bx + c = 0$:



$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

Fig. 3. Tree-like representation of an expression

The convenience of such tree-like structures is that they can be easily modified on the sub-tree level. Any sub-tree can be taken out and replaced with another one, preserving syntax validity of the expression.
However, the trees such as shown in Fig. 3 should be encoded in some computer-readable form for actual GP implementation. This can be done in a number of ways. Some systems (e.g. MATLAB) provide built-in mechanisms for storage and operation on hierarchical structures. If this is not available, string representations are employed. An expression or a program can be encoded in a common for imperative languages way; for example, the formula for the root of a square equation from Fig. 3 can be written as

$$\left(sqrt\left(b*b-4*a*c\right)-b\right)/\left(2*a\right) \tag{6}$$

Unfortunately, such representation, although being mathematically readable, is inconvenient to handle in a GP way. It has to be parsed to the tree-like form for every operation. Therefore, another syntax is traditionally used.
One can note that in the trees such as the ones above, an operation always precedes its arguments on the branch, e.g. instead of 'a + b' it reads '+ a b'. This notation is known as

prefix notation or Polish notation. It is used in the programming language LISP and its derivatives — certainly not the most human-friendly language but very flexible and useful in many areas, and is one of the most popular languages in GP field.

There are extensions to the tree-based GP. Most of them employ decomposition of the programs into sub-trees (modules) and evolving these modules separately. One of the most widely used methods of this kind is Koza's Automatically Defined Functions (ADF) (Koza, 1994). In ADF approach, the program is split into a main tree and one or more separate trees which take arguments and can be called by the main program or each other. In another approach, code fragments from successful program trees are automatically extracted and are held in a library, and then can be reused in the following generations by any individual via library calls.

However, tree-based GP is not the only option. It is possible to express a program as a linear sequence of commands. One of the examples of linear GP systems is stack-based GP (Perkins, 1994). In stack-based languages (such as Forth) each program instruction takes its arguments from a stack, performs its calculations and then pushes the result back onto the stack. For example, the sequence *1 2 +* . pushes the constants *1* and *2* onto the stack, then '+' takes these values from the stack, performs the addition and pushes the result *3* back. The final dot extracts and prints out the result. The notation such as '*1 2 +*' is the opposite to that used in LISP and is called *reverse Polish notation* (or *postfix notation*).

2.5.1.1 *Function set and terminal set*

When designing a GP implementation, proper care should be taken for choosing the function and terminal sets. The function set $F = \{f_1, f_2, \ldots, f_{nf}\}$ is the set of functions from which all the programs are built. Likewise, the terminal set $T = \{a_1, a_2, \ldots, a_{nt}\}$ consists of the variables available for functions. In principle, the terminals can be considered as functions with zero arguments and both the sets can be combined in one set of primitives $C = F \cup T$.

The choice of an appropriate set of functions and variables is crucial for successful solution of a particular problem. Of course, this task is highly problem dependent and requires significant insight. In some cases, it is known in advance that a certain set is sufficient to express the solution to the problem at hand.

However, in most practical real-world problems the sufficient set of functions and terminals is unknown. In these cases, usually all or most of the available data is supplied to the algorithm or iterative design is employed when additional data and functions are added if the current solution is unsatisfactory. As a result, the set of primitives is often far from the minimal sufficient set.

The effect of adding extraneous functions is complex. On the one hand, an excessive number of primitives may degrade performance of the algorithm, similar to choosing excessive genome length in GA. On the other hand, a particular additional function or variable may dramatically improve performance of both the algorithm and solution for a particular problem. For example, addition of the integral of error $\int (H - H_{set})dt$ as an input to altitude hold autopilot allows to eliminate static error and improve overall performance. Alternatively, the integrator function may be introduced along with both the current altitude reading H and the desired altitude H_{set}.

2.5.2 Initial population

Generation of the initial population in GP is not as straightforward as it usually is in conventional GAs. It has been noted above that the shape of a tree has (statistically) an