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).

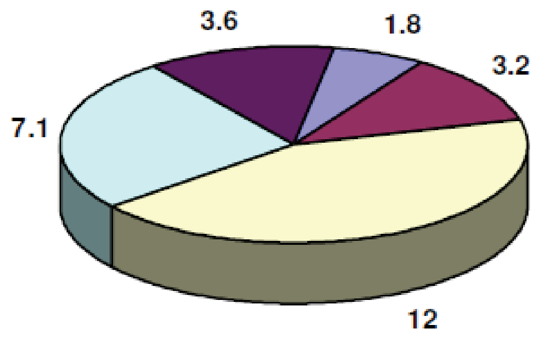

Fig. 1. Roulette wheel selection

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 $x_1 < x_2$. 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.

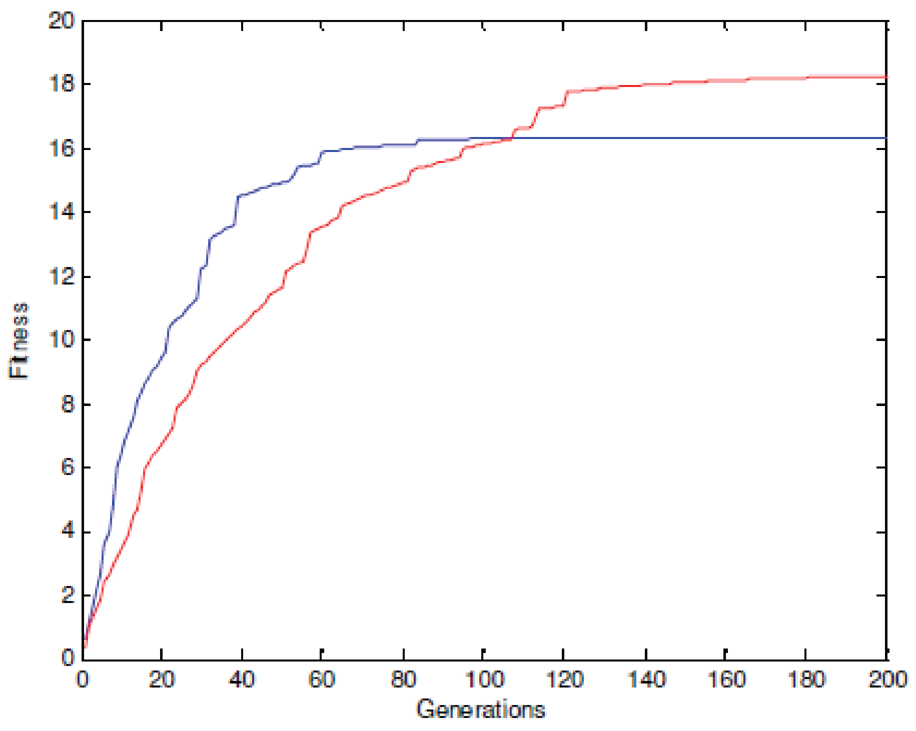

Fig. 2. Typical convergence graphs

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 $ax^2 + bx + c = 0$:

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

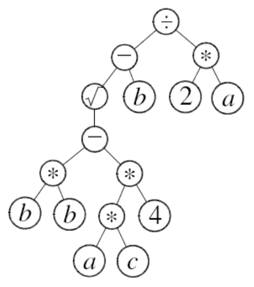

Fig. 3. Tree-like representation of an expression

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

influence on its evolution and that both sparse and bushy trees should be presented in the initial population. To this end, a so called 'ramped half-and-half' method, suggested by Koza (Koza, 1992), is typically used in GP. In this method, half of the trees in the population are generated as full trees and another half as random trees.
The 'ramped half-and-half' method employs two techniques for random tree generation: the 'full' method and the 'grow' method. Both of them start from choosing one of the functions from the function set F at random. It becomes the root of the new tree. Then, for each of the inputs of this function, a new primitive is selected with uniform probability. If the path from the root to the current point is shorter than the specified maximum depth, the new primitive is selected from the function set *F* for the 'full' method and from the union set *C* for the 'grow' method. If the path length reaches the specified depth, a terminal from the set *T* is selected at random for both methods. The process continues until the tree is complete, i.e. all the inputs are connected.

2.5.3 Genetic operators

2.5.3.1 *Crossover*

Crossover is usually the most important genetic operator in GP. Its classic variation (Koza, 1992) produces two children trees from two parent trees by exchanging randomly selected sub-trees of each parent. Both parents are selected using one of the stochastic selection methods such as fitness proportional selection and tournament selection. The crossover operation begins by choosing, using a uniform probability distribution, one random point in each parent independently to be the crossover point for that parent. The point may be a node as well as a leaf. Then, the sub-trees that have roots at the crossover points are removed from the parents, and the sub-tree from the second parent is inserted in place of the removed sub-tree of the first parent.
In terms of *S*-expressions, the sub-tree crossover is equivalent to exchanging the sublists of the parental lists. Considering the example from Fig. 3, the parental solutions are

$$\left(-(*\quad x\quad x)\quad \left(*(*\quad x\quad y)\ 2\right)\right) \text{ and } \left(/\quad \left(+\quad (*\quad x\quad x)\quad y\right)\quad 2\right) \tag{7}$$

The sub-lists corresponding to the selected crossover fragments are emphasized. These sublists are swapped between the parents and the following offspring are produced:

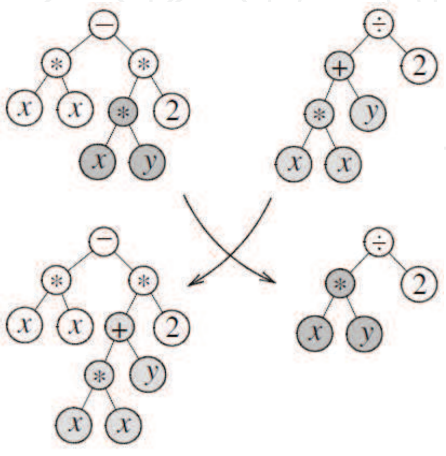

Fig. 4. Sub-tree crossover: $x^2 - 2xy$ crossed with $(x^2 + y)/2$ to produce $x^2 - 2(x^2 + y)$ and $xy/2$

$$\left(-\left(*\quad x\quad x\right)\quad\left(*\left(+\quad\left(*\quad x\quad x\right)\quad y\right)2\right)\right)\text{ and }\left(/\quad\left(*\quad x\quad y\right)\quad 2\right)\tag{8}$$

It can be noted that in such a simple operation, syntactic validity of the resulting expressions is always preserved.
To avoid excessive growth of the branches of the program trees, a maximum depth value is usually established. If the crossover operation produces an offspring of impermissible depth, this offspring is disregarded and its parent is reproduced as is. Koza (Koza, 1992) uses the default maximum depth of *17*. Even such a modest value allows creation of the trees of enormous size, up to $2^{17} - 1 = 131071$ (for binary trees).

2.5.3.2 *Other genetic operators*

Reproduction as such is simply copying of the selected individual into the next generation population. In classical GP (Koza, 1992), about *10%* of the population is selected for simple reproduction and *90%* is reproduced through crossover.
Mutation is typically a replacement of a randomly selected sub-tree of an individual with a new randomly generated sub-tree. A special case is point mutation, when a single random terminal is inserted in place of a sub-tree. In fact, point mutations sometimes happen during crossover operation. In general, mutation is less needed for GP than for GAs, because crossover alone can reintroduce genetic diversity. In many practical applications, mutation is not used at all.
Permutation is changing of the order of arguments of a randomly selected function. The effect and usefulness of permutation is roughly the same as that of mutation.
Editing is changing the shape and structure of a tree while maintaining its semantic meaning. Usually this implies a mathematical simplification of the expression. For example, the sub-trees which can be immediately evaluated may be replaced with a corresponding terminal, e.g. the expression (+ *2 3*) may be replaced with (*5*).
Simplification of the solutions may reduce (sometimes significantly) their length and thus reduce the computation time needed for their evaluation. In addition, shorter expressions are less vulnerable to disruption caused by crossover and mutation as there are fewer chances that they will be torn apart.

2.5.4 Convergence in GP

The question of convergence in GP is more complicated than in GAs. Generally, it is assumed in EA theory that the population converges when it contains substantially similar individuals (Langdon & Poli, 1992). Unlike conventional GAs, which have one-to-one mapping between genotype and phenotype, this rarely happens in GP. The search space in GP is essentially bigger than the phenotypic search space of the problem at hand. Any solution to the problem may be represented in an infinite number of ways.
Therefore, the population in GP may contain significantly different individuals (in terms of size and shape) and continue to evolve while yielding practically similar solutions to the problem.
However, the genotype cluster does not stabilise and continues to evolve. Since then, most of the highly fit individuals are produced by adding relatively insignificant branches to the successful core that came from the common ancestor. Therefore, each descendant genotype tends to be bigger than its parents. This results in a progressive increase in size known as bloat.

2.5.5 Bloat

The rapid growth of programs produced by GP is known since the beginning (Koza, 1992). As already noted, this growth need not to be correlated with increase of fitness because it consists of the code that does not change the semantics of the evolving programs. The rate of growth varies depending upon the particular GP paradigm being used, but usually exponential rates are observed (Nordin & Banzhaf, 1995).

For greater detail of different theories of bloat the reader is referred to (Langton, 1999) or (Langton & Poli, 2002). Generally, it should be noted that GP crossover by itself does not change the average program size. Bloat arises from the interaction of genetic operators and selection, i.e. selection pressure is required.

The most commonly (if not always) used restrictive technique is size and depth limits. Its implementation is already described in Section 2.5.3.1. It should be noted that the actual experiments (Langton & Poli, 2002) indicate that the populations are quickly affected by even apparently generous limits. Another commonly used approach is to give some preference to smaller solutions. The 'penalty' for excessive length may be included in fitness evaluation. However, in order not to degrade the performance of the algorithm, this component should be small enough so that it would have effect only for the solutions with identical phenotypic performance.

3. Aircraft flight control

Not unlike the generic control approach, aircraft flight control is built around a feedback concept. Its basic scheme is shown in Fig. 5. The controller is fed by the difference between the commanded reference signal *r* and the system output *y*. It generates the system control inputs *u* according to one or another algorithm.

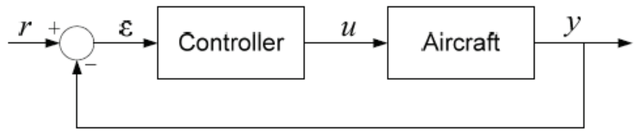

Fig. 5. Feedback system (one degree of freedom)

One of the main tasks of flight control as an engineering discipline is design of the controllers which enable a given aircraft to complete a defined mission in the most optimal manner, where optimality is based on mission objective.

3.1 Intelligent control

Intelligent control is a general and somewhat bold term that describes a diverse collection of relatively novel and non-traditional control techniques based on the *soft computing* approach. These include neural networks, fuzzy logic, adaptive control, genetic algorithms and several others. Often they are combined with each other as well as with more traditional methods.

Evolutionary and *genetic algorithms* (EAs, GAs) are global optimisation techniques applicable to a broad area of engineering problems. They can be used to optimise the parameters of various control systems, from simple PID controllers (Zein-Sabatto & Zheng, 1997) to fuzzy logic and neural network driven controllers (Bourmistrova, 2001; Kaise & Fujimoto, 1999). Another common design approach is evolutionary optimisation of trajectories, accompanied by a suitable tracking controller (e.g. (Wang & Zalzala, 1996)). An elaborated study of applications of EAs to control and system identification problems can be found in (Uzrem, 2003). As discussed above, Genetic Algorithms (in the form of *Genetic Programming*) are able to evolve not only the parameters, but also the *structure* of the controller.

In general, EAs require substantial computational power and thus are more suitable for offline optimisation. However, online evolutionary-based controllers have also been successfully designed and used. The *model predictive control* is typically employed for this purpose, where the controller constantly evolves (or refines) control laws using an integrated simulation model of the controlled system. A comprehensive description of this approach is given in (Onnen et al., 1997).

3.2 Flight control for the UAV recovery task

Aircraft control at recovery stage of flight can be conventionally separated into two closely related, but distinctive tasks: guidance and flight control. Guidance is the high-level ('outer loop') control intended to accomplish a defined mission. This may be path following, target tracking or various navigation tasks. Flight control is aimed at providing the most suitable conditions for guidance by maintaining a range of flight parameters at their optimal levels and delivering the best possible handling characteristics.

The landing of an aircraft is a well established procedure which involves following a predefined flight path. More often than not, this is a rectilinear trajectory on which the aircraft can be stabilised, and the control interventions are needed only to compensate disturbances and other sources of errors.

The position errors with respect to the ideal glidepath can be measured relatively easily. Shipboard landing on air carriers has tight error tolerances and absence of flare manoeuvres before touchdown. The periodic ship motion does have an effect on touchdown; however, it does not affect significantly the glidepath, which is projected assuming the average deck position.

Ship oscillations in high sea cause periodic displacement of the recovery window makes it impossible to project an optimal flight path when the final approach starts. Therefore, it turns out that the UAV recovery problem resembles that of homing guidance rather than typical landing. While stabilisation on a known steady flight path can be done relatively easy with a PID controller, homing guidance to a moving target often requires a more sophisticated control.

3.3 UAV controller structure

The objective is to synthesise such guidance strategy that enables reliable UAV recovery, and to produce a controller that implements the target tracking guidance strategy.

The evolutionary design (ED) method applied for this task allows to evolve automatically both the structure and the parameters of the control laws, thus potentially enabling to generate a 'full' controller, which links available measurements directly with the aircraft control inputs (throttle, ailerons, rudder and elevator) and implements both the guidance strategy and flight control (Fig. 6):

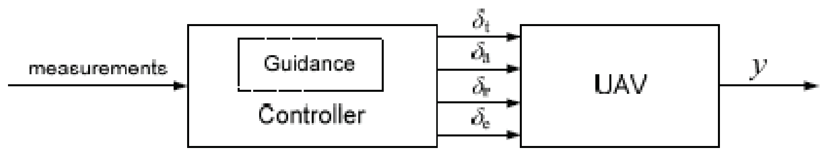

Fig. 6. Full controller with embedded guidance strategy

However, this approach, even though appealing at first and requiring minimum initial knowledge, proves to be impractical as the computational demands of the evolutionary algorithms (EAs) soar exponentially with the dimensionality of the problem. It is therefore desirable to reduce complexity of the problem by reducing the number of inputs/outputs and limiting, if appropriate, possible structures of the controllers.

Also it is highly desirable to decompose the complex control task into several simpler problems and to solve them separately. A natural way of such decomposition is separating the trajectory control (guidance) and flight control. The guidance controller issues commands u_g to the flight controller, which executes these commands by manipulating the control surfaces of the UAV (Fig. 7). These two controllers can be synthesised separately using appropriate fitness evaluation for each case.

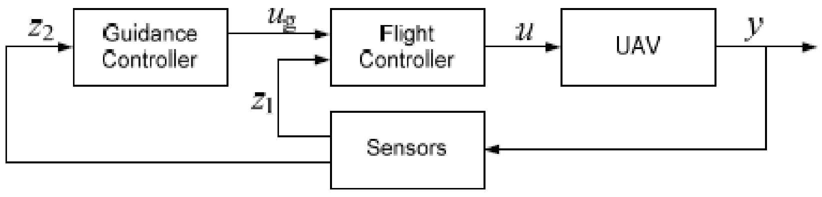

Fig. 7. UAV recovery control diagram

The internal structure of the controller is defined by the automatic evolutionary design based on predefined set of inputs and outputs. As an additional requirement is that the outputs u_g should represent a group of measurable flight parameters such as body accelerations, velocities and Euler angles, which the flight controller can easily track.

The structure of the output part of the guidance controller is as shown in Fig. 8 which allows to evolve the horizontal and vertical guidance laws separately, which may be desirable due to different dynamics of the UAV's longitudinal and lateral motion and also due to computational limitations.

Input measurements to the guidance controller should be those relevant to trajectory like positioning information, pitch and yaw angles and airspeed.

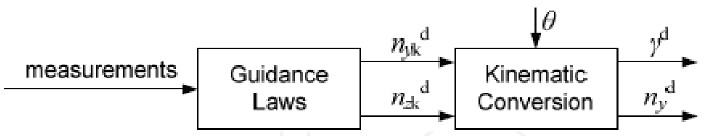

Fig. 8. Guidance controller

The derived quantities from raw measurements are the vertical and lateral velocity components with respect to the approach ground reference frame.

The system is based on radio distance metering and provides ten independent raw measurements (Fig. 9): three distances d_1, d_2 and d_3 from the UAV to the radio transmitters located at both ends of the recovery boom which supports the arresting wire and at the base of recovery mast; three rates of change of these distances; distance differences ($d_1 - d_2$) and ($d_3 - d_2$); and rates of change of the differences.

The guidance laws evolution process is potentially capable to produce the laws directly from raw measurements, automatically finding necessary relationships between the provided data and the required output.

Flight controller receives two inputs from the guidance controller: bank angle demand γ^d and normal body load factor demand n_y^d. It should track these inputs as precisely as possible by manipulating four aircraft controls: throttle, ailerons, rudder and elevator.

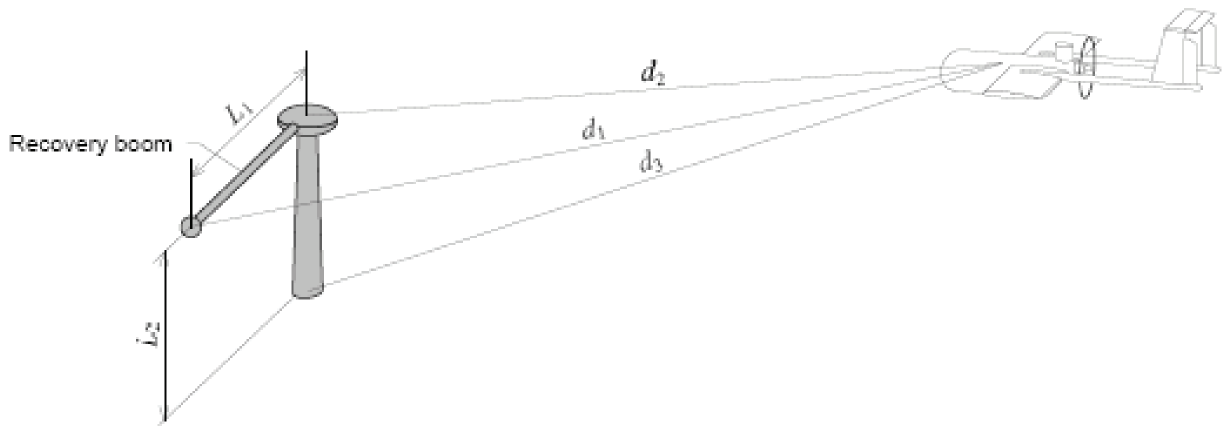

Fig. 9. Positioning scheme

The available measurements from the onboard sensors are body angular rates ω_x, ω_y, ω_z from rate gyros, Euler angles γ, ψ, θ, body accelerations n_x, n_y, n_z from the respective accelerometers, airspeed V_a, aerial angles a and β, actual deflection of the control surfaces δ_a, δ_r, δ_e, and engine rotation speed N_{rpm}.

For simplicity of design, the controller is subdivided into independent longitudinal and lateral components. In longitudinal branch, elevator tracks the input signal n_y^d, while throttle is responsible for maintaining a required airspeed. In lateral control, naturally, ailerons track γ^d, while rudder minimises sideforce by keeping n_z near zero.

4. Evolutionary Design

The Evolutionary Design (ED) presented in this section, generally, takes no assumptions regarding the system and thus can be used for wide variety of problems, including nonlinear systems with unknown structure. The core of evolutionary design is a specially tailored evolutionary algorithm (EA) which evolves both the structure and parameters of the control laws.

Parallel evolution can be implemented in a variety of ways. One of the few successfully employed variants is the *block structure controller evolution* (Koza et al., 2000).

In this work the ED algorithm enables to evolve suitable control laws within a reasonable time by utilising gradual evolution with the principle of strong casualty. This means that structure alterations are performed so that the information gained so far in the structure of the control law is preserved. Addition of a new block, though being random, does not cause disruption to the structure. Instead, it adds a new dimension and new potential which may evolve later during numerical optimisation.

The addition of new points or blocks is carried out as a separate dedicated operation (unlike sporadic structure alterations in the sub-tree crossover), and is termed *structure mutation*. Furthermore, in this work structure mutation is performed in a way known as *neutral structure mutation*. That's when the new block should be placed initially with zero coefficient. The usefulness of neutral mutations has been demonstrated for the evolution of digital circuits (Van Laarhoven & Aarts, 1987) and aerodynamic shapes (Olhofer et al., 2001). As a result, the ED algorithm basically represents a numerical EA with the inclusion of structure mutations mechanism.

Control laws are represented as a combination of static functions and input signals, which are organised as a dynamic structure of *state equations* and *output equations* in form of continuous representation.
The controller being evolved has *m* inputs, *r* outputs and *n* states. So, the controller comprises of *n* state equations and *r* output equations:

$$\begin{aligned} \dot{x}_1 &= g_1(x,u) \\ \dot{x}_2 &= g_2(x,u) \\ &\vdots \\ \dot{x}_n &= g_n(x,u) \end{aligned} \quad \text{and} \quad \begin{aligned} y_1 &= f_1(x,u) \\ y_2 &= f_2(x,u) \\ &\vdots \\ y_n &= f_n(x,u) \end{aligned} \tag{9}$$

where $\boldsymbol{u}$ is size *m* vector of input signals, $\boldsymbol{x} = [x_1,x_2,\ldots x_n]$ is size *n* vector of state variables, $y_{1\ldots r}$ are controller outputs. Initial value of all state variables is zero. All $n+r$ equations are built on the same principle and are evolved simultaneously. For structure mutations, a random equation is selected from this pool and mutated.
Input signals delivered to each particular controller are directly measured signals as well as the quantities derived from them. Within each group, inputs are organised in the subgroups of 'compatible' parameters. Compatible parameters are those which have close relationship with each other, have the same dimensions and similarly scaled. The examples of compatible parameters are the pairs (n_x, n_{xg}), (ω_y, $\dot{\psi}$), (V_a, V_{CL}).
Therefore, every controller input may be represented by a unique *code* consisting of three indices: the number of group *a*, the number of subgroup *b* and the number of item in the subgroup *c*. The code is designated as u(a,b,c).
Each of the control equations (9) is encoded as described above. To this end, only one single output equation of the form $y = f(\boldsymbol{u})$ will be considered in this section.
The encoding should allow a simple way to insert a new parameter in any place of the equation without disrupting its validity and in a way that this insertion initially does not affect the result, thus allowing neutral structure mutations. Conceptually, the equation is a sum of input signals, in which:

- every input is multiplied by a numeric coefficient or another similarly constructed expression;
- the product of the input and its coefficient (whether numeric or expression) is raised to the power assigned to the input;
- a free (absolute) term is present.

The simplest possible expression is a constant:

$$y = k_0 \tag{10}$$

A linear combination of inputs plus a free term is also a valid expression:

$$y = k_2u_2 + k_1u_1 + k_0 \tag{11}$$

Any numeric constant can be replaced with another expression. An example of a full featured equation is

$$y = ((k_4u_4 + k_3)u_3)^{-0.5} + k_2u_2 + (k_1u_1)^2 + k_0 \tag{12}$$

This algorithm can be illustrated by encoding the example (12). The respective internal representation of this expression is:

- Equation: $y = ((k_4u_4 + k_3)u_3)–0.5 + k_2u_2 + (k_1u_1)^2 + k_0$
- Expression: { u(3,-0.5) u(4) 1 2 u(2) 3 u(1,2) 4 5 }
- Object parameters: [k_4 k_3 k_2 k_1 k_0]
- Strategy parameters: [s_4 s_3 s_2 s_1 s_0]

This syntax somewhat resembles Polish notation with implicit '+' and '*' operators before each variable. The representation ensures presence of a free term in any sub-expression, such as k_3 and k_0 in the example above.

The algorithm of structure mutation is presented below.

1. Select an input (or a state variable) at random: u(a,b,c).
2. Obtain the initial values of the numeric coefficient and the strategy parameter
3. Append the object parameters vector with the initial coefficient, and the strategy parameters vector with the initial step size.
4. Form a sub-expression consisting of the selected variable code and the obtained index: { u(a,b,c) n }.
5. With 40% probability, set the insertion point (locus) to 1; otherwise, select a numeric value in the expression at random with equal probability among all numeric values present and set the locus to the index of this value.
6. Insert the sub-expression into the original expression at the chosen locus (*before* the item pointed).

This procedure may produce redundant expressions when the selected variable already exists at the same level. Thus an algebraic simplification procedure is implemented.

Fitness evaluation of a controller can be divided into two main stages. First is the preparation of the sample task and simulation of the model with the controller in the loop. The second stage is analysis of the results obtained from the simulation and evaluation of the fitness as such.

Other parameters of flight taken into account is control usage (or control effort): it is desirable to keep control usage at minimum.

The total fitness value is calculated as the weighted sum of all estimates:

$$F = W_cC_c + W_fC_f + W_eC_e + \dots \tag{13}$$

The exact value of the weighting coefficients W, as well as the number of estimates taken into account, is individual for each controller. As a rule, the weighting coefficients are chosen empirically.

Several steps of design algorithm need further clarification.

Initial population is initialised with the control laws of the form $y = const$ with randomly chosen constants.

The *selection* is performed deterministically. The populations used in this study were of moderate size, usually 24 to 49 members. Selection pressure is determined by the number of the offspring n of each individual. The smaller n, the lower the selection pressure. For nearly all runs in this work n = 2, which means that half of the population is selected. This is a rather mild level of selection pressure.

The parameters determining *structure mutation* occurrence, k_s and P_s, both change during the evolution. The probability of structure mutation P_s is normally high in the beginning (0.7 to 1.0) and then decrease exponentially to moderate levels (0.4 to 0.6) with the exponent 0.97 to the generation number. In the beginning of evolution, k_s is reduced by half until the generation 20 and 100 respectively.

Reproduction is performed simultaneously with mutation, as it is typically done in ES, with the exception that this operation is performed separately for each selected member.

5. Controller synthesis and testing

The UAV control system is synthesised in several steps. First, the flight controller is produced. This requires several stages, since the flight controller is designed separately for longitudinal and lateral channels. When the flight controller is obtained, the guidance control laws are evolved.

Application of the ED algorithm to control laws evolution is fairly straightforward: 1) preparation of the sample task for the controller, 2) execution of the simulation model for the given sample task and 3) analysis of the obtained performance and evaluation of the fitness value. For a greater detail of control system design reader is referred to (Bourmistrova & Khantsis, 2009). When both the model and fitness evaluation are prepared, the final evolution may be started. Typically, the algorithm is run for 100–200 generations (depending on complexity of the controller being evolved). The convergence and the resulting design is then analysed and the evolution, if necessary, is continued.

Step 1. *PID autothrottle* - Initially a simple PID variant of autothrottle is evolved - to ensure a more or less accurate airspeed hold. At the next stage, its evolution is continued in a full form together with the elevator control law. The PID structure of the controller may be ensured by appropriate initialisation of the initial population and by disabling structure mutations. Therefore, the algorithm works as a numerical optimisation procedure. The structure of the autothrottle control law is following:

$$\begin{aligned}\dot{x}_1 &= k_1 x_1 + k_2 \Delta V_a \\ \delta_t &= k_3 x_1 + k_4 \Delta V_a + k_5 \dot{V}_a + k_6\end{aligned} \tag{14}$$

where $\Delta V_a = V^d - V_a$ is the airspeed error signal, δ_t is the throttle position command and $k_{1\ldots6}$ are the coefficients to be optimised.

Step 2. *Longitudinal control* - With a simple autothrottle available, elevator control can be developed to provide tracking of the normal body load factor demand n_y^d. Altogether, the reference input signal is

$$n_y^d(t) = 0.03\left(H^d - H(t)\right) + \cos\theta(t) + 0.2c(t) \tag{15}$$

Elevator control law is initialised as follows:

$$\begin{aligned}\dot{x}_2 &= 0 \\ \delta_e &= k_1 x_2 + k_2 \Delta n_y + k_3\end{aligned} \tag{16}$$

where $\Delta n_y = n_y^d - n_y$ is the load factor error and the coefficients $k_{1\ldots3}$ are sampled at random for the initial population.

Step 3. *Lateral control* - Lateral control consists of two channels: ailerons control and rudder control. As a rule, for an aerodynamically stable aircraft such as the *Ariel* UAV, lateral control is fairly simple and is not as vital for flight as longitudinal control. For this reason, both control laws, for ailerons and rudder, are evolved simultaneously in one step.

Both ailerons and rudder control laws are initialised in a similar manner to (16):

$$\begin{aligned} \dot{x}_3 &= 0 \\ \dot{x}_4 &= 0 \\ \delta_a &= k_{11}x_3 + k_{12}\Delta\gamma + k_{13} \\ \delta_r &= k_{21}x_4 + k_{22}n_z + k_{23} \end{aligned} \tag{17}$$

Step 4. *Guidance* - At this point, flight controller synthesis is completed and guidance laws can be evolved. Guidance controller comprises two control laws, for the vertical and horizontal load factor demands, n_{yk}^d and n_{zk}^d respectively. This is equivalent to acceleration demands

$$a_{yk}^d = gn_{yk}^d \text{ and } a_{zk}^d = gn_{zk}^d \tag{18}$$

These demands are passed through the kinematic converter to form the flight controller inputs n_y^d and γ^d .

In the guidance task, the random factors include initial state of the UAV; Sea State, atmospheric parameters, and initial state of the ship.

Fitness is calculated as follows:

$$F_g = 40\Delta h_1^2 + 20\Delta z_1^2 + 50|\psi_1| + 25|\gamma_1| + 500C_c\left(n_{yk}^d\right) + 500C_c\left(n_{zk}^d\right) + 100C_f\left(n_{zk}^d\right) \tag{19}$$

where Δh_1 and Δz_1 are vertical and horizontal miss distances, and ψ_1 and γ_1 are final yaw and bank angles. Greater weight for vertical miss than that for horizontal miss is used because vertical miss allowance is smaller (approximately 3–4 m vs. 5 m) and also because vertical miss may result in a crash into the boom if the approach is too low.

Algorithm initialisation is the same as for longitudinal flight control laws evolution, except that elitism is not used and the population size is 48 members. The initial population is sampled with the control laws of the form

$$\begin{aligned} \dot{x}_6 &= 0 \\ \dot{x}_7 &= 0 \\ a_{yk}^d &= k_{11}x_6 + k_{12} \\ a_{zk}^d &= k_{21}x_7 + k_{22} \end{aligned} \tag{20}$$

where all coefficients *k* are chosen at random. For convenience, the control laws are expressed in terms of accelerations, which are then converted to load factors according to (18). Since these control laws effectively represent 'empty' laws *y* = *const* (plus a low-pass output filter with randomly chosen bandwidth), structure mutation is applied to each member of the initial population.

From the final populations, the best solution is identified by calculating fitness of each member using *N* = 25 simulation runs, taking into account also success rate. As an eample the best performing controller is the following (unused state variables are removed) with 100% success rate in the 25 test runs with average fitness 69.52:

$$\begin{aligned}\dot{x}_6 &= -3.548\omega_v \\ a_{yk}^d &= 0.916x_6 - 54.3\omega_v - 0.1261 \\ a_{zk}^d &= -107.68\omega_h + 0.0534V_{zT} + 0.4756\end{aligned} \tag{25}$$

The other attempted approach is the pure proportional navigation (PPN) law for recovery which was compared with the obtained solutions (Duflos et al., 1999; Siouris & Leros, 1988). The best PN controller have been selected using the fitness values of each member of the final population averaged over 100 simulation runs. It is the following:

$$\begin{aligned}a_{yk}^d &= -3.27V_{CLa}\omega_V \\ a_{zk}^d &= -3.18V_{CLa}\omega_h\end{aligned} \tag{26}$$

This controller received the fitness 88.85 and showed success rate 95%.

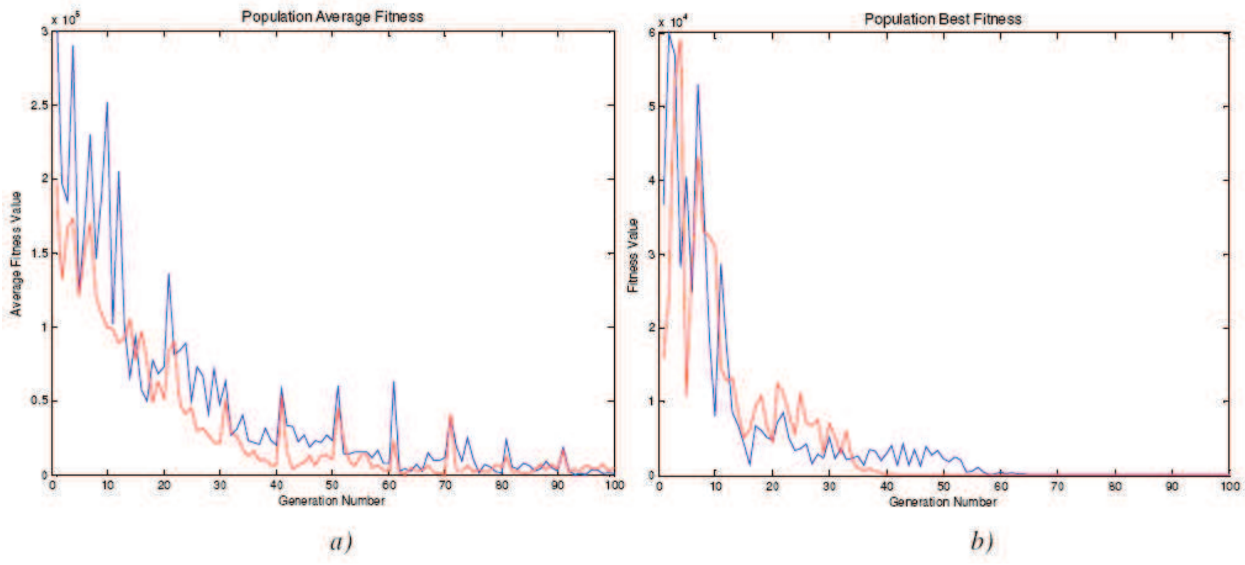

Fig. 12. Population average fitness (a) and best fitness (b) for two independent evolutions of guidance controller

5.1 Controller testing

In this work, two main types of simulation tests are conducted – robustness and performance. Results are presented for robustness test, which is aimed at ensuring the controller has good robustness to modelling uncertainties and to test whether the controller is sensitive to specific perturbations.

Perturbations were introduced in two ways - in form of variation of physical quantities and by introducing additional dynamic elements and by changing internal variables such as aerodynamic coefficients. For a realistic test, all perturbations should be applied simultaneously to identify the worst case scenario. However, single perturbation tests (the *sensitivity analysis*) allow to analyse the degree of influence of each parameter and help to plan the robustness test more systematically.

In this type of test, a single model variable is perturbed by a set amount and the effect upon the performance of the controller is determined. Performance evaluation can be measured in a manner similar to fitness evaluation of the guidance controller (equation (19)).

The additional parameters taken into account in performance measurement are impact speed V_{imp} and minimum altitude $H_{\min}$ attained during the approach.
Altogether, the performance cost (PC) is calculated as follows:

$$PC = 40\Delta h_1^2 + 20\Delta z_1^2 + 50|\psi_1| + 25|\gamma_1| + 50 f_V + 20 f_H + 10C_c(\delta_a) + 10C_c(\delta_r) + C_c(\delta_e) + \dots \\ + 200C_f(\delta_a) + 200C_f(\delta_r) + 200C_f(\delta_e) \tag{21}$$

where

$$f_V = \begin{cases} V_{imp} - 30, & V_{imp} > 30 \\ 0, & V_{imp} \le 30 \end{cases}, \quad f_H = \begin{cases} 10 - H_{\min}, & H_{\min} > 10 \\ 0, & H_{\min} \le 10 \end{cases}. \tag{22}$$

Impact speed V_{imp} and minimum altitude $H_{\min}$ are measured in m/s and metres respectively. Other designations are as in (19). Unlike fitness evaluation in the flight controller evolution, the commanded control deflections δ_a, δ_r and δ_e are saturated as required for control actuators.
The environment delivers a great deal of uncertainty. The range of disturbances and the initial ship phase are chosen to provide a moderately conservative estimation of controller performance. Results for different dynamic scenarios are presented in (Bourmistrova & Khantsis, 2009).
The parameters corresponding to aircraft geometry and configuration are tested in a similar manner. The range is increased to the scale factors between 0 and 10 (0 to 1000%) with step 0.05. The allowable perturbations (as factors to the original values) are summarised in Table 1, where * denotes the extreme value tested.

Parameter	Lower limit factor		Upper limit factor	
	Controller 1	Controller 2	Controller 1	Controller 2
Empty mass (m_{empt})	0.50*	0.50*	1.54	1.52
Rolling moment of inertia (I_x)	0.20	0.20	10*	2.43
Yawing moment of inertia (I_y)	0*	0*	10*	10*
Pitching moment of inertia (I_z)	0*	0*	2.17	1.91
xy cross product of inertia (I_{xy})	0*	0*	10*	10*
Wing area (S)	0.74	0.87	1.55	1.60

Table 1. Allowable perturbations of UAV inertial properties and geometry

Example in Fig. 13 demonstrates results for varying empty mass. Dashed cyan and green lines on the trajectory graphs represent the 'unrolled' along the flight path traces of the tips of the recovery boom (lateral position on the top view and vertical position on the side view). The bar on the right hand side illustrates the size of recovery window.
For some parameters, no noticeable drop in performance is experienced within the testing limits. These limits indicate quite large perturbations that can reasonably be expected. The main perturbations that cause a performance degradation are those that affect the physically attainable trajectory and not the operation of the controller, e.g. increasing weight and decreasing wing area. However, as follows from the above analysis, careful adjustment of elevator efficiency and/or horizontal stabiliser incidence may help to increase the tolerance to increased weight. There is still sufficient margin in angles of attack and engine power.

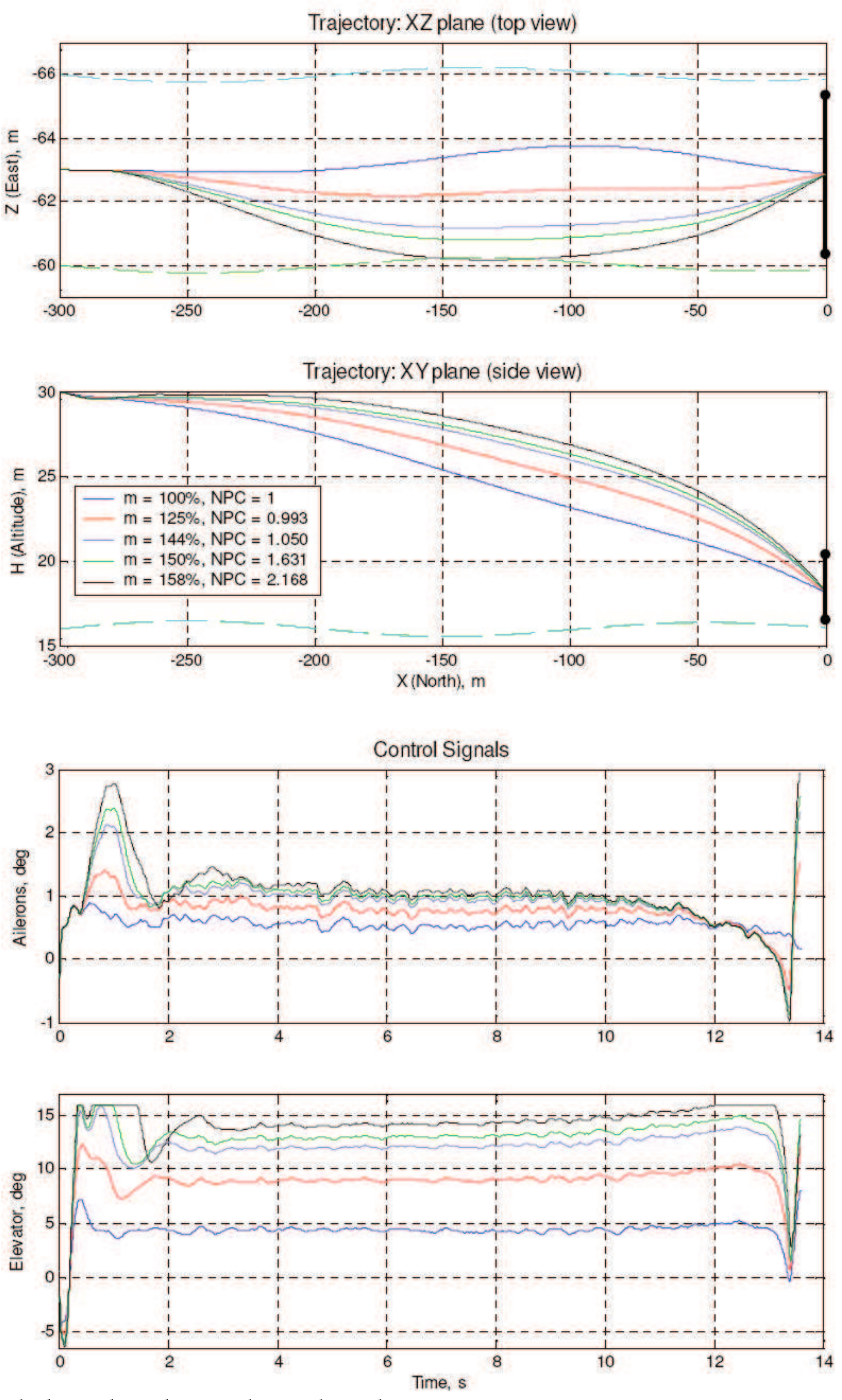

Fig. 13. Flight path and control signals with varying empty mass

The other parameters evaluated in this research were the perturbations of the power unit parameters, the aircraft aerodynamics parameters and sensor noise (Khantsis, 2006).
Overall, these tests have not exposed any significant robustness problems within the controllers. Large variations in single aircraft parameters caused very few control problems. However, such variations cannot realistically judge the performance of the controllers under simultaneous perturbations of multiple parameters.

6. Conclusions

In this chapter, an application of the Evolutionary Design (ED) is demonstrated. The aim of the design was to develop a controller which provides recovery of a fixed-wing UAV onto a ship under the full range of disturbances and uncertainties that are present in the real world environment.
Evolutionary computation is an attractive and quickly developing technique. Methodological shortcomings of the early approaches and lack of intercommunication between different EA schools contributed to the fact that evolutionary computation remained relatively unknown to the engineering and scientific audience for almost three decades. Despite computational demands, evolutionary methods offer many advantages over conventional optimisation and problem solving techniques. The most significant one is flexibility and adaptability of EAs to the task at hand.
In this study, a combination of the EA methods is used to evolve a capable UAV recovery controller. An adapted GP approach is used to represent the control laws. However, these laws are modified more judiciously (yet stochastically) than commonly accepted in GP and evolved in a manner similar to ES approach.
One of the greatest advantages of developed methodology is that minimum or no a priori knowledge about the control methods is used, with the synthesis starting from the most basic proportional control or even from 'null' control laws. During the evolution, more complex and capable laws emerge automatically. As the resulting control laws demonstrate, evolution does not tend to produce parsimonious solutions.
The method demonstrating remarkable robustness in terms of convergence indicating that a near optimal solution can be found. In very limited cases, however, it may take too long time for the evolution to discover the core of a potentially optimal solution, and the process does not converge. More often than not, this hints at a poor choice of the algorithm parameters.
The simulation testing covers the entire operational envelope and highlights several conditions under which recovery is risky. All environmental factors—sea wave, wind speed and turbulence—have been found to have a significant effect upon the probability of success. Combinations of several factors may result in very unfavourable conditions, even if each factor alone may not lead to a failure. For example, winds up to *12* m/s do not affect the recovery in a calm sea, and a severe ship motion corresponding to Sea State 5 also does not represent a serious threat in low winds. At the same time, strong winds in a high Sea State may be hazardous for the aircraft.
On the whole, Evolutionary Design is a useful and powerful tool for complex nonlinear control design. Unlike most other design methodologies, it tries to *solve* the problem at hand automatically, not merely to optimise a given structure. Although ED does not exclude necessity of a thorough testing, it can provide a near optimal solution if the whole range of conditions is taken into account in the fitness evaluation. In principle, no specific knowledge

about the system is required, and the controllers can be considered as 'black boxes' whose internals are unimportant. Successful design of the controller for such a challenging task as shipboard recovery demonstrates great potential abilities of this novel technique.

7. References

Beyer, H.-G. & Deb, K. (2001). On self-adaptive features in real-parameter evolutionary algorithms, Proceedings on *IEEE Transactions on Evolutionary Computation*, pp. 250-270 vol. 5(3), ISBN 1089-778X, Germany, June 2001

Bourmistrova, A. (2001). Knowledge based control system design for autonomous flight vehicle, *PhD thesis*, RMIT University: Melbourne, September 2001

Bourmistrova, A. & Khantsis, S. (2009). Flight control system design optimisation via Genetic Programming, in *Recent advances in signal processing*, IN-TECH, ISBN: 978-953-7619-41-1

Chipperfield, A. & Fleming, P. (1996). Systems integration using evolutionary algorithms, Proceedings of *UKACC International Conference on Control '96*, pp. 705- 710 vol.1, October 1996

Coley, D. A. (1998). *An Introduction to Genetic Algorithms for Scientists and Engineers.* University of Exeter, ISBN 9810236026, 9789810236021, Published by World Scientific, 1999

K. A. De Jong, W. M. Spears. *On the state of evolutionary computation*, ed. S. Forrest, in *5th International Conference on Genetic Algorithms* proceedings. San Mateo, CA, Morgan Kaufmann: p. 618, 1993.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs, Proceedings on *International Conference on Genetic Algorithms and their Applications*, pp. 183- 187, ed. J. J. Grefenstette, Carnegie-Mellon University, Pittsburgh May 1985

Deb, K., Joshi, D. & Anand, A. (2001). Real-coded evolutionary algorithms with parentcentric recombination, *KanGAL Report 2001003*, Indian Institute of Technology.

Duflos, E., Penel, P. & Vanheeghe, P. (1999). 3D guidance law modelling, IEEE Transactions on Aerospace and Electronic Systems, Vol. 35, No 1, page numbers (72-83), (January 1999), ISSN: 0018-9251

Eshelman, L. J. & Schaffer, J. D. (1993). Real-coded genetic algorithms and interval schemata, in *Foundations of Genetic Algorithms 2*, page numbers (187-202), ed. L. D. Whitley, Morgan Kaufmann: San Mateo, CA, 1993

Fogel, D. B. (1962). Autonomous automata, *Industrial Research*, Vol. 4, page numbers (14-19), 1962

Fogel, D. B., Owens, A. J. & Walsh, M. J. (1966). *Artificial intelligence through simulated evolution.* New York: Wiley, 1966

Goldberg, D. E. (1989). *Genetic Algorithms in search, optimisation and machine learning.* Addison-Wesley, 1989

Goldberg, D. E. & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms, Foundations of Genetic Algorithms, page numbers (69-93), ed. G. J. E. Rawlins. Morgan Kaufmann, San Mateo, CA, ISBN-10: 1558601708

Grefenstette, J. J. (1999). Evolvability in dynamic fitness landscapes: A genetic algorithm approach, Proceedings on *Congress of Evolutionary Computation*, IEEE Press, pp. 2031–2038 vol. 3, Washington, ISBN: 0-7803-5536-9

Herrera, F., Lozano, M. & Verdegay, J. L. (1998). Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis, *Artificial Intelligence Review,* page numbers (265-319), Vol. 12, No 4, Springer, ISSN 0269-2821

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press

Hollstien, R. B. (1971). Artificial genetic adaptation in computer control systems, *Doctoral dissertation*, University of Michigan

Kaise, N. & Fujimoto, Y. (1999). Applying the evolutionary neural networks with genetic algorithms to control a rolling inverted pendulum, *Lecture Notes in Computer Science : Simulated Evolution and Learning,* Volume 1585/1999, (January 1999), page numbers (223-230), Springer Berlin / Heidelberg, ISBN 978-3-540-65907-5

Khantsis, S., (2006). Control System Design Using Evolutionary Algorithms for Autonomous Shipboard Recovery of Unmanned Aerial Vehicles, *PhD thesis*, RMIT University: Melbourne, 1992

Koza, J. R. (1992). *Genetic Programming: On the programming of computers by means of natural selection,* Cambridge, Massachusetts: MIT Press, December 1992 (6th ed. 1998), ISBN-10:0-262-11170-5

Koza, J. R. (1994). *Genetic Programming II: Automatic discovery of reusable programs.* Cambridge, Massachusetts: MIT Press, May 1994, ISBN-10:0-262-11189-6

Koza, J. R., Bennett III, F. H., Andre D. & Keane, M. A. (1996). Four problems for which a computer program evolved by genetic programming is competitive with human performance, Proceedings on *IEEE International Conference on Evolutionary Computation*, Nagoya, Japan, May 1996, ISBN: 0-7803-2902-3

Koza, J. R., Bennett III, F. H., Andre, D. & Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving.* Kluwer Academic Publishers, ISBN 1-55860-543-6

Koza, J. R., Keane, M. A., Yu, J., Bennett III, F. H. & Mydlowec, W. (2000). Automatic creation of human-competitive programs and controllers by means of genetic programming, *Genetic Programming and Evolvable Machines*, Vol. 1, No 1, page numbers (121-164), (January 2000), ISSN: 1389-2576

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J. & Lanza G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers

Langdon, W. B (1999). The evolution of size and shape, in *Advances in genetic programming 3*, ed. L. Spector, W. B. Langdon and et al, MIT Press, page numbers (163-190), July 1999, ISBN-10:0-262-19423-6

Langdon, W. B. & Poli, R. (2002). *Foundations of genetic programming.* Berlin: Springer-Verlag, ISBN 978-3-540-42451-2

Mahfoud, S. W. & Mani, G. (1996). Financial forecasting using genetic algorithms. *Applied Artificial Intelligence*, page numbers (543-565), Vol 10, November 1996, ISSN: 1087-6545

McFarland, R. E. & Duisenberg, K. (1995). Simulation of rotor blade element turbulence, *Technical Memorandum 108862*, Acc. No. NASA TM-108862, NASA, January 1995

McLean, D. & Matsuda, H. (1998). Helicopter station-keeping: comparing LQR, fuzzy-logic and neural-net controllers, *Engineering Applications of Artificial Intelligence*, page numbers (411- 41811), Vol. 11, November 1998, ISSN 0952-1976

Nordin, P. & Banzhaf, W. (1995). Complexity compression and evolution, ed. L. J. Eshelman, Proceedings on *6th International Conference on Genetic Algorithms*, pp. 310-317, Pittsburgh, PA, Morgan Kaufmann, July 1995

Olhofer, M., Jin, Y. & Sendhoff, B. (2001). Adaptive encoding for aerodynamic shape optimization using Evolution Strategies, Proceedings of *Congress on Evolutionary Computation*, pp. 576-583 vol. 1, Seoul, South Korea, 2001, ISBN:1-59593-010-8

Onnen, C., Babuska, R., Kaymak, U., Sousa, J. M., Verbruggen, H. B. & Isermann, R. (1997). Genetic algorithms for optimization in predictive control, *Control Engineering Practice*, Vol. 5, No 10, page numbers (1363-1372), (October 1997), ISSN: 0967-0661

Ono & Kobayashi, S. (1997). A real-coded genetic algorithm for function optimisation using unimodal normal distribution crossover, ed. T. B$ck, Proceedings on *7th International Conference on Genetic Algorithms*, pp. 246-253, San Mateo, CA, Morgan Kauffman, 1997

Perkins, T. (1994). Stack-based genetic programming, Proceedings on *IEEE World Congress on Computational Intelligence*, pp. 148-153 vol 1, Orlando, Florida, IEEE Press, 27-29 June 1994

Schwefel, H.-P. (1981). *Numerical optimisation of computer models.* New York: Wiley

Sendhoff, B. & Kreuz, M. (1999). *Variable encoding of modular neural networks for time series prediction*, ed. V. W. Porto, Proceedings on *Congress on Evolutionary Computation*, pp. 259-266, IEEE Press, Piscataway, NJ, July 1999

Siouris, G. M., Leros, P. (1988). Minimum-time intercept guidance for tactical missiles, *Control Theory and Advanced Technology*, Vol. 4, No 2, page numbers (251-263), (February, 1988), ISSN 0911-0704

Smith, S. F. (1980). A learning system based on genetic adaptive algorithms, *PhD dissertation*, University of Pittsburgh

Spears, W. M. & Anand, V. (1991). A study of crossover operators in genetic programming, ed. Z. W. Ras, M. Zemankova, Proceedings on *International Symposium on Methodologies for Intelligent Systems*, pp. 409-418, Berlin: Springer-Verlag, 542., 1991, ISBN:3-540-54563-8

Ursem, R. K. (2003). Models for evolutionary algorithms and their applications in system identification and control optimization, *PhD dissertation*, Department of Computer Science, University of Aarhus, Denmark, 2003

van Laarhoven, P. J. M. & Aarts, E. H. L. (1987). *Simulated annealing: theory and applications.* Dordrecht, The Netherlands: D. Reidel, ISBN-10: 9027725136

Wang, Q. & Zalzala, A. M. S. (1996). Genetic control of near time-optimal motion for an industrial robot arm, Proceedings of *IEEE International Conference on Robotics and Automation*, pp. 2592-2597 vol. 3, ISBN: 0-7803-2988-0, Minneapolis, MN, April 1996