EVOLUTIONARY IDENTIFICATION OF DYNAMICAL SYSTEMS

IVAN ZELINKA
ROMAN SENKERIK
ZUZANA OPLATKOVA

Faculty of Applied Informatics
Department of Applied Informatics
Tomas Bata University in Zlin
Nad Stranemi 4511 Zlin, Czech Republic
Email: {zelinka,senkerik,oplatkova}@fai.utb.cz

Abstract: Synthesis, identification and control of the complex dynamical systems are usually extremely complicated. When classics methods are used, some simplifications are required which tends to lead to idealized solutions that are far away to reality. In contrast, the class of methods based on evolutionary principles is successfully used to solve this kind of problems with a high level of precision. In this paper a novel method is discussed which has been successfully proven in many simulations like neural network synthesis and electrical circuit synthesis. Typical examples are discussed, alongside the use of evolutionary algorithms on dynamical system synthesis - identification.

Keywords: symbolic regression, genetic programming, grammar evolution, analytic programming, SOMA

1 INTRODUCTION

The term *symbolic regression* (SR) represents a process, in which measured data is fitted by a suitable mathematical formula such as $x^2 + C$, $\sin(x)+1/e^x$, etc., Mathematically, this process is quite well known and can be used when data of an unknown process is obtained. Historically SR has been in the preview of manual manipulation, however during the recent past, a large inroad has been made through the use of computers. Generally, there are two well-known methods, which can be used for SR by means of computers. The first one is called genetic programming or GP [Koza, 1998; Koza *et al.*, 1999] and the other is grammatical evolution [O'Neill and Ryan, 2002; Ryan *et al.*, 1998].

The idea as to how to solve various problems using SR by means of evolutionary algorithms (EA) was introduced by John Koza who used genetic algorithms (GA) for GP. Genetic programming is basically a symbolic regression, which is done by using of evolutionary algorithms instead of a human brain. The ability to solve very difficult problems is now well established, and hence, GP today performs so well that it can be applied, e.g. to synthesize highly sophisticated electronic circuits [Koza et al. 2003].

In the last decade of the 20th century, C. Ryan developed a novel method for SR, called grammatical evolution (GE). Grammatical evolution can be regarded as an unfolding of GP because of some common principles, which are the same for both algorithms. One important characteristic of GE is that it can be implemented in any arbitrary computer language compared with GP, which is usually done (in its canonical form) in LISP. In contrast to other evolutionary algorithms, GE was used only with a few search strategies, with a binary representation of the populations [O'Sullivan and Ryan, 2002]. Another interesting investigation using symbolic regression was carried out by [Johnson 2003] working on Artificial Immune Systems or/and systems which are not using tree structures like linear genetic programming (full text is at https://eldorado.uni-dortmund.de/bitstream/2003/20098/2/Brameierunt.pdf) and another similar algorithm to AP, Multi Expression Programming (see http://www.mep.cs.ubbcluj.ro/).

In this paper, a different method is presented which is called analytic programming (AP) [Zelinka 2002 a], [Zelinka 2002 b], [Zelinka and Oplatkova, 2003], [Zelinka and Oplatkova, 2004], [Zelinka, Chen, Celikovski, 2008]. AP is also a tool for symbolic regression, based on different principles compared to GP and GE. The important principles of AP, together with tests and a comparison of its main principles with GP and GE, are presented.

2 MOTIVATION

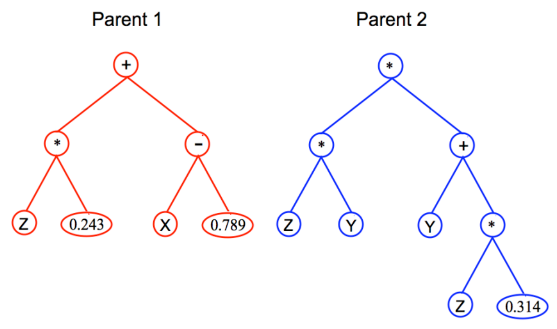
The motivation to develop analytic programming was based on several facts. In the case of GP, individuals consist of various functions and terminals [Koza 1998]. For example, when an offspring is created, the principles of GA are used and two chromozomes are cut into two parts, which are then exchanged. It means that the main principles of GA are a part of GP. When the same philosophy is followed in the case of different algorithms, for example in differential evolution (DE), [Price 1999], incorrect functional structures would be obtained. This is due to the fact that the offspring creation in DE is based on arithmetical operations. Their usage ensures that when a new individual (offspring - trial vector) is created via a noisy vector, then the nth parameter can look like "F*(cos()-tan()) + mod()" after direct application of DE rules. It is clear that there is no direct way to create a program based on such kinds of operations.

A similar situation of GP exists for GE, because the individual is also represented in the binary string and a new offspring is created in a similar way as in GP (crossover, mutation). Despite this fact, there is one theoretical possibility as to how GE can be used by an arbitrary evolutionary algorithm and some investigations were made in [O'Sullivan and Ryan 2002]. Because there is no possible use of GP and GE in their fundamental forms by arbitrary evolutionary algorithm, it would be suitable to develop some universal method for symbolic regression. A method, showing attributes of such universality was developed during 2000-2003 and it is called *analytic programming* (AP). The remaining part of this paper is focused on its brief description of existing methods and simulations on selected case examples.

3 GENETIC PROGRAMMING AND GRAMMATICAL EVOLUTION

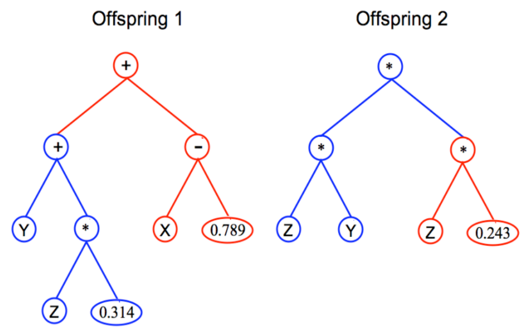
The term *genetic programming* (GP) was coined by J.R. Koza (see for example [Koza 1998]). Main principle of GP is based on genetic algorithm which is working with a population of individuals, represented in LISP programming language. Individuals, in its canonical form of GP, are not binary strings as is the case for GA, but consist of LISP symbolic objects like sin(), +, Exp(), MyFunction, etc. The origin of these objects are from LISP or they are simply user defined functions. Symbolic objects are usually divided into two classes: functions and terminals. Functions were just explained and terminals represent a set of independent variables like x, y, and constants like π, 3.56, etc here.

The main principle of GP is usually demonstrated by means of so-called trees (basically graphs with nodes and edges, Figures 1 and 2) representing individuals in LISP symbolic syntax. Individual in the shape of a tree, or formula like *0.234 z + x –* 0.789 are called *programs*. Because GP is based on GA, then evolutionary steps (mutation, crossover, …) in GP are in principle the same as in GA. As an example of GP, assume two artificial parents – trees in Figure 1 and 2 representing programs *0.234 z + x – 0.789* and *z y (y+0.314 z)*. When a crossover is applied, then, for example, subset of trees are exchanged. The resulting offspring's are given in Figure 2.



$$Z*0.234+X-0.789 \qquad Z*Y*(Y+Z*0.314)$$

Figure 1: Parental program in genetic programming



$$(Y+Z*0.314)+X-0.789 \qquad Z*Y*(Z*0.243)$$

Figure 2: Offspring program in genetic programming

Subsequently, the offspring's fitness is calculated so that behaviour of just synthesized and evaluated individual-tree should ideally be similar to the desired behavior. The term *desired behaviour* can be understood as a measured data set from some process (program should fit them as well as possible) or like optimal robot trajectory, i.e. program is a realization of a sequence of robot commands (Move_Left, Stop, Go_Forward,…) leading to the final position as ideally as possible. This is basically the same for GE and AP. Due to the complex and rich theory of GP, it is recommended for detailed description of GP to see [Koza 1998], [Koza 1999].

Another method doing the same procedure, from the point of view of the resulting program like GP was developed [O'Neill, Ryan 2002] and is called *grammatical evolution*.

Grammatical evolution is in its canonical form based on GA, however thanks to a few important changes it has a few advantages when compared to GP. The main difference is individual coding. Whereas GP manipulates in LISP symbolic expressions, GE uses individuals, based on binary strings. These are transformed into integer sequences and mapped into final program by means of so called Backus-Naur form (BNF) by the following artificial example. Assume that T = {+,-,*,/, X, Y} is a set of operators and terminals and F = {epr, op, var}, the so called *nonterminals*. In this case, grammar used for final programme synthesis is given by Table 1. Rules used for individual transforming into a program is based on (1). Grammatical evolution is based on binary chromosome with variable length, divided into so called *codons* (range of integer value 0-255), which is transformed into a integer domain according to Table 2.

$$\begin{array}{l}\text{unfolding} = \text{codon mod rules} \\ \text{where rules is number of rules for given nonterminal}\end{array} \quad (1)$$

Table 1: GE grammar in artificial example

Nonterminals		*Unfolding*	*Index*
expr	::=	op expr expr	(0)
		var	(1)
op	::=	+	(0')
		-	(1')
		*	(2')
		/	(3')
var	::=	X	(0'')
		Y	(1'')

Table 2: Chromozome transformation

Chromozome	**Binary**	**Integer**	**BNF index**
Codon 1	00101000	40	(0)
Codon 2	11000011	162	(2')
Codon 3	00001100	67	(1)
Codon 4	10100010	12	(0'')
Codon 5	01111101	125	(1)
Codon 6	11100111	231	(1'')
Codon 7	10010010	146	Unused
Codon 8	10001011	139	Unused

Synthesis of actual programme is as follows: Start begins with nonterminal object *expr*. Because integer value of Codon 1 (see Table 2) is 40 then according to (1) is unfolding of *expr* = *op expr expr* (40 mod 2, 2 rules for *expr,* i.e. (0) and (1)). Consequently, Codon 2 is used for unfolding of *op* by * (162 mod 4) which is terminal and thus unfolding for this part of program is closed. Then it continues in unfolding of the remaining nonterminals (*expr expr*) till the final program is fully closed by terminals. If the program is closed before the end of the chromosome is reached, then the remaining codons are ignored. Otherwise it continues again from the beginning of the chromosome. The final program is based on the described example is in this case X*Y (see Figure 3). For fully detailed description of GE principles, it is recommended to see [O'Neill, Ryan 2002].

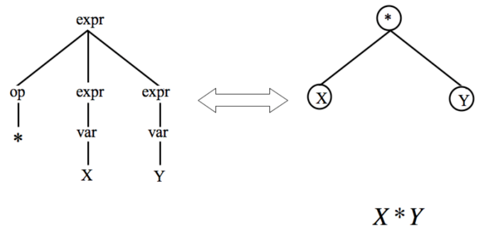


Figure 3: Tree structure of final program.

3 ANALYTIC PROGRAMMING

3.1 General Ideas

Very similar introduction is in [Zelinka, Oplatkova, Nolle, 2005], however for readers comfort we have decided to partly repeat rewritten introduction of AP here.

The term *analytic programming* was coined by the authors of this article as a matter of simplicity: Since it is possible to use almost any evolutionary algorithm (computationally verified) for AP, each EA used for the new approach would add its name to the emerging algorithm, e.g. SOMA [Zelinka 2004] programming, DE programming, SA programming etc. This clearly would be confusing and complicated. Analytic programming indicates the use of an EA for analytic solutions synthesis (i.e. symbolic regression), thus it was the main reason for choosing the term 'analytic programming'.

Analytic programming was inspired by the numerical methods in Hilbert functional spaces and by GP. The principles of AP lie somewhere between these two philosophies: From GP stems the idea of the evolutionary creation of symbolic solutions, whereas the general ideas of functional spaces and the building of resulting function by means of a search process (usually done by numerical methods such as the Ritz or Galerkin method) are adopted from Hilbert spaces. Like GP or GE, AP is based on a set of functions, operators and so-called terminals, which are usually constants or independent variables, for example:

- functions: Sin, Tan, Tanh, And, Or
- operators: +, -, *, /, dt,…
- terminals: 2.73, 3.14, t,…

All these 'mathematical' objects create a set from which AP tries to synthesize an appropriate solution. The main principle of AP is based on discrete set handling (DSH), proposed in [Zelinka 2004] (see Figure 4). Discrete set handling itself can be seen as a universal interface between EA and the problem to be solved symbolically. That is why AP can be carried out using almost any evolutionary algorithm. Analytical programming, together with a few basic examples, is discussed in more detail in [Zelinka and Oplatkova, 2003].

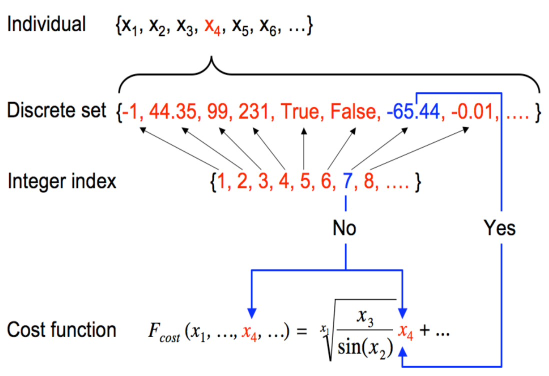

Individual
{x1, x2, x3, x4, x5, x6, ...}
Discrete set
{-1, 44.35, 99, 231, True, False, -65.44, -0.01, ... }
Integer index
{1, 2, 3, 4, 5, 6, 7, 8, ... }
No
Yes
Cost function
$F_{cost}(x_1, \ldots, x_4, \ldots) = \sqrt[x_1]{\frac{x_3}{\sin(x_2)}}\, x_4 + \ldots$

Figure 4: Discrete set handling.

The set of mathematical objects are functions, operators and so-called terminals (usually constants or independent variables). All these objects are mixed together as shown in Figure 5 and consists of functions with different number of arguments. Because of the variability of the content of this set, it is called for article purposes, "general functional set" – GFS. The structure of GFS is nested i.e. it is created by subsets of functions according to the number of their arguments. The content of GFS is dependent only on the user. Various functions and terminals can be mixed together. For example GFS$_{all}$ is a set of all functions, operators and terminals, GFS$_{3arg}$ is a subset containing functions with only three arguments, GFS$_{0arg}$ represents only terminals, etc.

GFS$_{all}$ = {+,-,/,^, Sin, Cos, Tan, t, x, Mod, …}
GFS$_{3arg}$ = {User_Function, Beta, …}
GFS$_{2arg}$ = {Log, Mod, Gamma, …}
GFS$_{1arg}$ = {Sin, Cos, Tan, Abs, …}
GFS$_{0arg}$ = {t, x, y, π, …}

Figure 5: Principle of the general functional set

Briefly, in AP, individuals consist of non-numerical expressions (operators, functions,…) as described above, which are in the evolutionary process represented by their integer indexes (Figure 6 and 7). This index then serves as a pointer into the set of expressions and AP uses it to synthesize the resulting function-program for cost function evaluation.

AP was tested in three versions. All three versions used for program synthesis the same sets of functions, terminals, etc., as Koza applied in GP [Koza 1998], [Koza et al 1999]. The second version (AP$_{meta}$, termed first version AP$_{basic}$) is modified in the sense of constant estimation. For example, Koza used for the so-called sextic problem [Koza 1998] randomly generated constants, whereas AP here uses only one, called K which is inserted into the formula (2) at various places, by the evolutionary process. When a program is synthesized, then all K's are indexed so that K_1, K_2, …, K_n, are obtained (3) in the formula, and then all K_n are estimated using a second evolutionary algorithm (4). Because EA (slave) "works under" EA (master, i.e. EA$_{master}$ ▶ program ▶ K indexing ▶ EA$_{slave}$ ▶ estimation of K$_n$) then this version is called AP with metaevolution - AP$_{meta}$.

$$\frac{x^2+K}{\pi^K} \tag{2}$$

$$\frac{x^2+K_1}{\pi^{K_2}} \tag{3}$$

$$\frac{x^2+3.56}{\pi^{-229}} \tag{4}$$

Because this version is quite time consuming, AP$_{meta}$ was modified to the third version, which differs from the second one in the estimation of K. This is done by using a suitable method for non-linear fitting (AP$_{nf}$). This method has shown the most promising performance when unknown constants are present. Results of some comparative simulations can be found in [Zelinka 2002 a], [Zelinka 2002 b], [Zelinka and Oplatkova 2003], [Zelinka and Oplatkova 2004], [Zelinka, Oplatkova and Nolle 2004]. For the simulations described here, AP$_{nf}$ was used.

3.2 Data Set Structure and Mapping Method

The subset structure presence in GFS is vitally important for AP. It is used to avoid synthesis of pathological programs, i.e. programs containing functions without arguments, etc. Performance of AP is, of course, improved if functions of GFS are expertly chosen based on experiences with solved problem.

The important part of the AP is a sequence of mathematical operations, which are used for program synthesis. These operations are used to transform an individual of a population into a suitable program. Mathematically said, it is mapping from an individual domain into a program domain. This mapping consists of two main parts. The first part is called discrete set handling (DSH) and the second one is security procedures, which do not allow to synthesize pathological programs. Discrete set handling proposed in [Lampinen, Zelinka 1999], [Zelinka 2004] is used to create an integer index, which is used in the evolutionary process like an alternate individual handled in EA by method of integer handling. The method of DSH, when used, allows to handle arbitrary objects including nonnumeric objects like linguistic terms {hot, cold, dark,…}, logic terms (True, False) or other user defined functions. In AP, DSH is used to map an individual into GFS and together with security procedures (SP) creates above mentioned mapping which transforms arbitrary individual into a program. Individuals in the population consist of integer parameters, i.e. an individual is an integer index pointing into GFS.

Analytic programming is basically a series of function mapping. Presented in Figure 6 and 7, is a demonstrated example of how a final function is created from an integer individual. Number 1 in the position of the first parameter means that the operator "+" from GFS_{all} is used (the end of the individual is far enough). Because the operator "+" has to have at least two arguments, the next two index pointers 6 (sin from GFS) and 7 (cos from GFS) are dedicated to this operator as its arguments. Both functions, sin and cos, are one-argument functions so the next unused pointers 8 (tan from GFS) and 9 (t from GFS) are dedicated to sin and cos function. Because as an argument of cos is used, variable t on this part of resulting function is closed (t is zero-argument) in its AP development. One-argument function tan remains and because there is one unused pointer 9, tan is mapped on "t" which is on 9th position in GFS.

To avoid synthesis of pathological functions, a few security "tricks" are used in AP. The first one is that GFS consists of subsets containing functions with the same number of arguments. Existence of this nested structure is used in the special security subroutine which measures as to how far the end of individual is and according to this, objects from different subsets are selected to avoid pathological function synthesis. If more arguments are desired than it is possible (the end of the individual is near), the function will be replaced by another function with the same index pointer from subset with lower number of arguments. For example, it can occur if the last argument for one argument function will not be terminal (zero-argument function) as demonstrated in Figure 7.

GFS need not be constructed only from clear mathematical functions as is demonstrated, but also other user-defined functions, which can be used, e.g. logical functions, functions which represent elements of electrical circuits or robot movement commands.

3.3 Crossover, Mutations and Other Evolutionary Operations

During the evolution of a population, different operators are used, such as crossover and mutation. In comparison with GP or GE, evolutionary operators like mutation, crossover, tournament, and selection are fully in the competence of the used evolutionary algorithm. Analytic programming does not contain them in any point of view in its internal structure. Analytic programming is created like superstructure of EAs for symbolic regression independent on their algorithmical structure. Operations used in evolutionary algorithms are not influenced by AP and vice versa. For example, if DE is used for symbolic regression in AP then all evolutionary operations are done according to the DE rules.

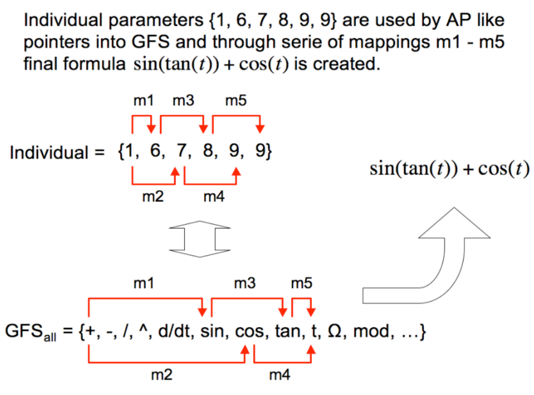

Individual parameters {1, 6, 7, 8, 9, 9} are used by AP like pointers into GFS and through serie of mappings m1 - m5 final formula sin(tan(t)) + cos(t) is created.
m1 m3 m5
Individual = {1, 6, 7, 8, 9, 9}
m2 m4
sin(tan(t)) + cos(t)
m1 m3 m5
GFS_all = {+, -, /, ^, d/dt, sin, cos, tan, t, Ω, mod, …}
m2 m4

Figure 6: Main and simplified principle of AP

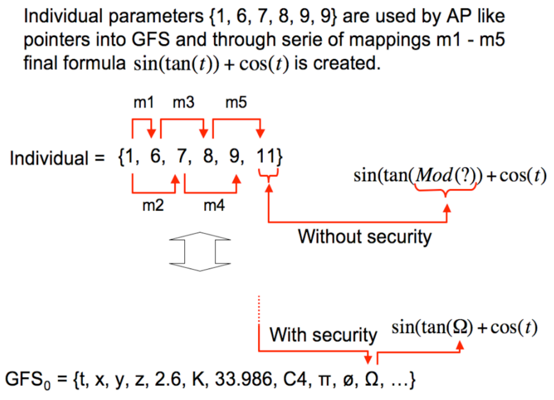

Individual parameters {1, 6, 7, 8, 9, 9} are used by AP like pointers into GFS and through serie of mappings m1 - m5 final formula sin(tan(t)) + cos(t) is created.
m1 m3 m5
Individual = {1, 6, 7, 8, 9, 11}
m2 m4
sin(tan(Mod(?)) + cos(t)
Without security
With security
sin(tan(Ω) + cos(t)
GFS_0 = {t, x, y, z, 2.6, K, 33.986, C4, π, ø, Ω, …}

Figure 7: AP principle with security principles. According to the main idea of AP, a Mod function would be selected as an argument for Tan function. When security subroutine measure how far the end of individual is, then Mod is replaced by Ω from the terminal subset.

3.4 Reinforced Evolution

During evolution, less or more suitable individuals are synthesized. Some of these individuals are used to reinforce evolution towards better solution synthesis. The main idea of reinforcement is based on addition of just synthesized and partly successful program into initial set of terminals. Reinforcement is based on user-defined threshold (temporarily acceptable cost value or fitness) used in the decision as to which individual will be used for addition into the initial set of terminals.

For example, if the threshold is set to 5, and fitness of all individuals - programs in the population is larger than 5, then the evolution is running on basic, i.e. initial GFS. When the best of all individuals in the actual population is less than 5, then it is completely added into the initial GFS and is marked like terminal. Henceforth, the evolution is running on enriched GFS containing partially successful program. Thanks to this fact, evolution is able to synthesize final solutions much faster than AP without reinforcement. This fact has be exhaustively verified through countless simulations on a number of different problems. When a program is added into GFS, then the threshold is also set to its fitness. If another individual with lower fitness is found, the threshold is synthesized and the older threshold is overwritten by the new value.

It is quite similar to automatically defined functions (ADF) from GP, however the set of functions and terminals in GP can contain more than one ADF (which of course, theoretically increase complexity of searched space according to N!) including properly defined arguments of these ADF and critical situations checking (selfcalling,…). This is not a problem of AP reinforcement, because the added program into initial GFS is regarded like a terminal (or terminal structure), i.e. no function, no arguments, no selfcalling, etc. and cardinality of initial GFS set increase only by one.

3.5 Security procedures

Security procedures (SP) included in AP as well as in GP, are used to avoid various critical situations. In the case of AP security procedures were not developed for AP purposes after all, but they are mostly an integrated part of AP. However, sometimes they have to be defined as a part of cost function, based on the kind of situation arising (for example situation 2, 3 and 4, see below). Critical situations are like:

1. pathological function (without arguments, self-looped...)
2. functions with imaginary or real part (if not expected))
3. infinity in functions (dividing by 0, …)
4. „frozen“ functions (an extremely long time to get a cost value - hrs...)
5. etc…

Simply, a SP can be regarded here as a mapping from an integer individual to the program where it is checked for how far the end of the individual is, and based on this information a sequence of mapping is redirected into a subset with lower number of arguments. This satisfies the condition that no pathological function will be generated. Other activities of SP are integrated parts of cost function to satisfy items 2-4, etc.

3.6 Similarities and Differences

Because AP was partly inspired by GP, then between AP, GP and GE some differences as well as some similarities logically are there. A few of these are:

- Synthesized programs (similarity): AP as well as GP and GE is able to do symbolic regression in a general point of view. It means that output of AP is according to all-important simulations [Zelinka 2002 a], [Zelinka 2002 b], [Zelinka and Oplatkova 2003], [Zelinka and Oplatkova 2004] and [Zelinka, Oplatkova and Nolle 2004] similar to programs from GP and GE (see www.ft.utb.cz/people/zelinka/ap).
- Functional set (similarity): AP_{basic} operates in principle, on the same set of terminals and functions as GP or GE while AP_{meta} or AP_{nf} use universal constant K (difference) which is indexed after program synthesis.
- Individual coding (difference): coding of an individual is different. Analytic programming uses an integer index instead of direct representation as in canonical GP. GE uses the binary representation of an individual, which is consequently converted into integers for mapping into programs by means of BNF [O'Neill and Ryan 2002].
- Individual mapping (difference): AP uses discrete set handling, [Zelinka 2004] while GP in its fundamental form uses direct representation in Lisp [Koza 1998] and GE uses Backus-Naur form (BNF).
- Constant handling (difference): GP uses a randomly generated subset of numbers – constants [Koza 1998], GE utilizes user determined constants and AP uses only one

constant K for AP_{meta} and AP_{nf}, which is estimated by other EA or by nonlinear fitting.

- Security procedures (difference): to guarantee synthesis of non-pathological functions, procedures are used in AP which redirect the flow of mapping into subsets of a whole set of functions and terminals according to the distance of the end of the individual. If pathological function is synthesized in GP, then synthesis is repeated. In the case of GE, when the end of an individual is reached, then mapping continues from the individual beginning, which is not the case of AP. It is designed so that a non-pathological program is synthesized before the end of the individual is reached (maximally when the end is reached).

3.7 Some selected programs

During AP's development and research simulations, various kinds of programs have been synthesized (see www.fai.utb.cz/people/zelinka/ap). In (5), (6) two mathematical programs are shown for final program complexity demonstration, which were randomly generated amongst 1000 programs to check if the final structure is free of pathologies – i.e. if all functions has the right number of arguments, etc. In this case, there was no paid attention on mathematical reasonability of following test programs based on clear mathematical functions.

$$\frac{\log\left(e^{\cosh\left(\cot\left(\frac{1}{\sqrt{\frac{t^{2.6}-1}{t^{2.6}+1}}\left(t^{2.6}+1\right)}\right)\right)}\right)}{\log\left(-\coth^{-1}\left(\sqrt[4]{-1}-\cosh^{-1}(i)\right)\right)} \tag{5}$$

$$\sqrt{t}\left(\frac{1}{\log(t)}\right)^{\sec^{-1}(1.28)} \log^{\sec^{-1}(1.28)}\left(\sinh\left(\sec\left(\cos(1)\right)\right)\right) \tag{6}$$

Other examples come from real comparative simulations with GP. Example (7) demonstrates one concerning the Boolean problems solution, (8) synthesized formula for polynomial fitting problem in general form (see general constant K after indexing, i.e. K[[1]]…) and (9) its exact and final state after nonlinear constants K estimation.

(7)

Nand[Nand[Nor[A,B],C&&B],Nor[Nor[C,A],C&& C&&A]]||Nand[Nand[Nand[C&&A,C&&A],C&&A &&A],Nand[C&&A||C&&A,A]]

$$x\left(K_1+\frac{\left(x^2K_3\right)}{K_4\left(K_5+K_6\right)}\right)* \left(-1+K_2+2x\left(-x-K_7\right)\right) \tag{8}$$

$$x\,(1.99 + 2(2.14^{-17} - x)\,x)(0.50 - 0.50x^2) \tag{9}$$

Interested readers can find more at AP website www.fai.utb.cz/people/zelinka/ap.

3.8 Analytic Programming - Summary

Summarily a different approach to the symbolic regression called analytic programming is described. Based on its results and structure, it can be stated that AP appears to be a universal candidate for symbolic regression by means of different search strategies. Problems on which AP was applied were selected from test and theory problems domain as well as from real life problems. They were:

- Random synthesis of function from GFS, 1000 times repeated. The aim of this simulation was to check if AP could generate pathological function. In this simulation, randomly generated individuals were created and consequently transformed into programs and checked for their internal structure. No pathological program was identified.
- sin(t) approximation, 100 times repeated. AP was used to synthesize program function sin(x) fitting.
- $\left|\left|\cos(t)\right|+\sin(t)\right|$ approximation, 100 times repeated, the same as in the previous example. The main aim was again the fitting of dataset generated by a given formula.
- Solving of ordinary differential equations (ODE): u''(t) = cos(t), u(0) = 1, u(π) = -1, u'(0) = 0, u'(π) = 0, 100 times repeated. AP was searching for suitable function, which would solve this case of ODE.
- Solving of ODE: ((4 + x)u''(x))'' + 600u(x) = 5000(x-x2), u(0)=0, u(1)=0, u''(0)=0, u''(1)=0, 5 times repeated (due to longer time of simulation in the Mathematica® environment). Again as in the previous case, AP was used to synthesize a suitable function – solution of this kind of ODE. This ODE was used from and represents a civil engineering problem in reality.

- Boolean even and symmetry problems according to the [Koza 1998] for comparative reasons.
- Simple neural network synthesis by means of AP – a simple few layered NN synthesis was tested by AP.
- Optimal robot trajectory estimation (56th kongres IAC2005, 17-21.10, Fukuoka Japan, see also www.iac2005.org).
- Controller synthesis – for a selected class of systems, a controller was synthesized.
- Deterministic chaos synthesis [Zelinka, Chen, Celikovsky, 2008]

Remaining text in this article is focused on dynamical system synthesis.

4 Dynamical System Identification

4.1 Problem selection

The class of selected dynamical systems has been selected for this case study:

$$\frac{1}{s^2 + 2s + 1} \qquad (10)$$

$$\frac{1}{(s+1)^3} \qquad (11)$$

$$\frac{1-0{,}5s}{s(s+1)} \qquad (12)$$

$$\frac{1}{(1+s)(1+0{,}5s)(1+0{,}5^2s)(1+0{,}5^3s)} \qquad (13)$$

Systems (10) – (13) are stable, excluding system (12) which is unstable. This system has been added here to check whether EAs are able identify also unstable systems and has been selected from a representative class of dynamical systems. Systems like those above selected are used in the control engineering for testing of identification algorithms. Due to this fact were selected for simulations in this paper.

4.2 The Fitness Function

The fitness (cost function) has been calculated using the distance between the behavior of the original system and the synthesized candidate program (14). The minimal value (the best solution) is 0 for all problems. The aim of all the simulations was to find the best solution, i.e. a solution that returns the cost value 0.

$$f_{\cos t} = \sum_{i=1}^{10} \left| OS_i - SS_i \right| \qquad (14)$$

OS_{i} - output in the ith iteration of the original system

SS_{i} - output in the ith iteration of the synthesized system

4.3 Optimisation Algorithm Used and Parameter Setting

For the experiments described here, stochastic optimisation algorithms, such as Differential Evolution, version DERand1Bin (DE) [Price 1999] and SelfOrganizing Migrating Algorithm, strategy All-To-One (SOMA) [Zelinka 2004], had been used.

Differential Evolution [Price, 1999] is a population-based optimization method that works on real-number coded individuals. For each individual $\mathbf{x}_{i,G}$ in the current generation *G*, DE generates a new trial individual $\mathbf{x'}_{i,G}$ by adding the weighted difference between two randomly selected individuals $\mathbf{x}_{r1,G}$ and $\mathbf{x}_{r2,G}$ to a third randomly selected individual $\mathbf{x}_{r3,G}$. The resulting individual $\mathbf{x'}_{i,G}$ is crossed-over with the original individual $\mathbf{x}_{i,G}$. The fitness of the resulting individual, referred to as a perturbated vector $\mathbf{u}_{i,G+1}$, is then compared with the fitness of $\mathbf{x}_{i,G}$. If the fitness of $\mathbf{u}_{i,G+1}$ is greater than the fitness of $\mathbf{x}_{i,G}$, $\mathbf{x}_{i,G}$ is replaced with $\mathbf{u}_{i,G+1}$, otherwise $\mathbf{x}_{i,G}$ remains in the population as $\mathbf{x}_{i,G+1}$. Differential Evolution is robust, fast, and effective with global optimization ability. It does not require that the objective function is differentiable, and it works with noisy, epistatic and time-dependent objective functions.

SOMA is a stochastic optimization algorithm that is modelled on the social behaviour of co-operating individuals [Zelinka, 2004]. It was chosen because it has been proved that the algorithm has the ability to converge towards the global optimum [Zelinka, 2004]. SOMA works on a population of candidate solutions in loops called *migration loops*. The population is initialized randomly distributed over the search space at the beginning of the search. In each loop, the population is evaluated and the solution with the highest fitness becomes the leader *L*. Apart from the leader, in one migration loop, all individuals will traverse the input space in the direction of the leader. Mutation, the random perturbation of individuals, is an important operation for evolutionary strategies (ES). It ensures the diversity amongst the individuals and it also provides the means to restore lost information in a population. Mutation is different in SOMA compared with other ES strategies. SOMA uses a parameter called PRT to achieve perturbation. This parameter has the same effect for SOMA as mutation has for GA.

The novelty of this approach is that the PRT Vector is created before an individual starts its journey over the search space. The PRT Vector defines the final movement of an active individual in search space.

The randomly generated binary perturbation vector controls the allowed dimensions for an individual. If an element of the perturbation vector is set to zero, then the individual is not allowed to change its position in the corresponding dimension. An individual will travel a certain distance (called the path length) towards the leader in *n* steps of defined length. If the path length is chosen to be greater than one, then the individual will overshoot the leader. This path is perturbed randomly.

The control parameter settings have been found empirically and are given in Table 5 (SOMA) and Table 6 (DE). The main criterion for this setting was to keep the same setting of parameters as much as possible and of course the same number of cost function evaluations as well as population size (parameter PopSize for SOMA, NP for DE). Individual length represents number of optimized parameters. For an exact description of the algorithms see [Price 1999] for DE and [Zelinka 2004] for SOMA.

Table 5: SOMA setting

PathLength	3
Step	0.11
PRT	0.1
PopSize	400
Migrations	100
MinDiv	-0.1
Individual Length	100

Table 6: DE setting

NP	400
F	0.8
CR	0.2
Generations	2000
Individual Length	100

5 EXPERIMENTAL RESULTS

Both algorithms (SOMA, DE) have been applied 20 times in order to find the optimal structure of the identified dynamical system. The primary aim of this comparative study is not to show which algorithm is better and worse, but to show that AP can be actually used for different problems of symbolic regression by different EAs also based on previous comparative studies and case studies [Zelinka 2002 a], [Zelinka 2002 b], [Zelinka and Oplatkova, 2003], [Zelinka and Oplatkova, 2004], [Zelinka, Oplatkova and Nolle 2004], and [Zelinka, Chen, Celikovsky, 2008]. Outputs of all simulations are depicted in Figures 8 - 13 and numerically reported in Tables 7. Histograms 8 – 11 report frequency of fitness based on successful simulations. It is important to note that system (13) at Figure. 11 has been successfully synthesized only 3 times. This is probably due to the structure of the cost function and is open for future research. Simulation for the remaining systems has been successful for all 20 simulations. In Figure 12 and 13 is a typical example of synthesized system response.

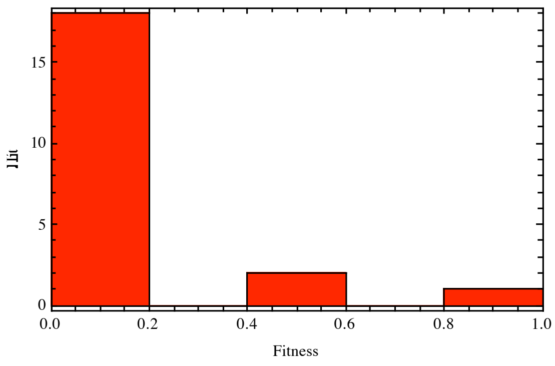


Figure 8 Histogram of system (10), randomly selected results are reported in (Table 7)

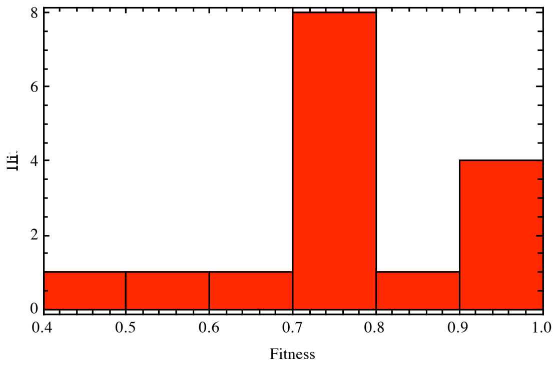


Figure 9. Histogram of system (11), randomly selected results are reported in (Table 7)

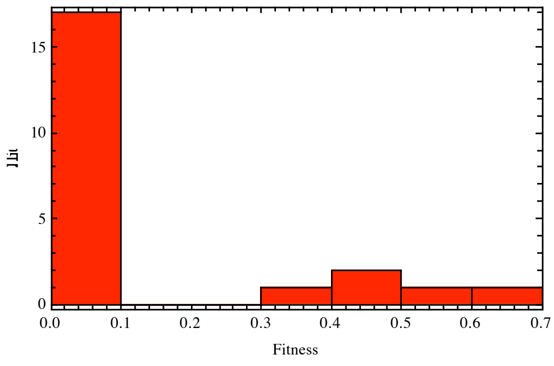


Figure 10. Histogram of system (12), randomly selected results are reported in (Table 7)

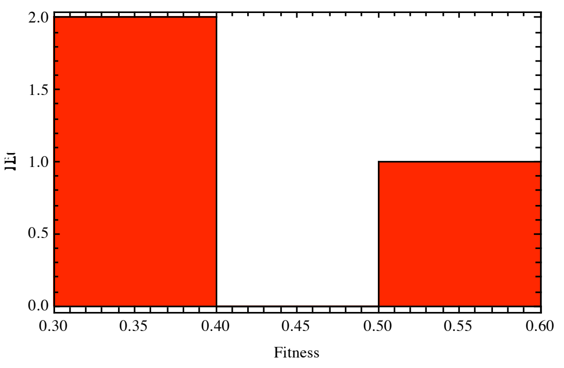


Figure 11. Histogram of system (13), randomly selected results are reported in (Table 7)

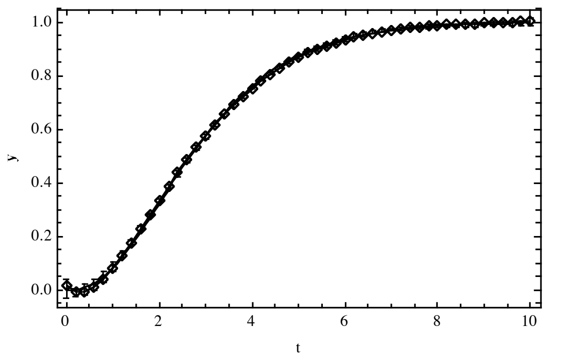


Figure 12. Typical synthesized system response for system (11).

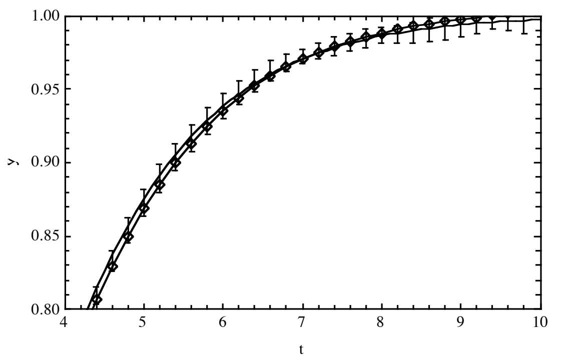


Figure 13. Zoom from Figure 12. Small bars with squares in the middle represents minimal, maximal and average values calculated from all simulations for studied system. Two lines are response of the original system and the best approximation by synthesized system.

The structure of the synthesized systems was found to be quite rich. As an example, reported here is the selected results for systems (11) by (15) (in general structure with nonestimated constants) and (16) (final form, estimated constants and converted to time domain).

$$\frac{K_1(1+s+K_3)(s+s^2-\frac{s(-s+K_5-K_6^2)}{K_4})}{(s-K_2)(\frac{s^2}{K_7}+2s(s+K_8))} \tag{15}$$

$$e^{-0.720807t}(-1+e^{-0.00008344t})-6331.11(-1.38717 (-1+e^{-0.72089t})-1.38733e^{-0.720807t}(-1+e^{-0.720807t}))- 0.00055\ \mathrm{UnitStep}[t] \tag{16}$$

6 CONCLUSIONS

The method of analytic programming described here is relatively simple, easy to implement and easy to use. Based on its principles and its universality (it was tested with 4 evolutionary algorithms – SA, GA, SOMA and DE in the past, see for example [Zelinka, Oplatkova, Nolle 2004] or [Zelinka, Chen, Celikovski, 2008] it can be stated that AP is a superstructure of an algorithm rather than an algorithm itself.

The main aim of this paper was to show how the structure of the various dynamical systems can be identified by means of evolutionary algorithms applied in AP. Analytic programming was used here in basic comparative simulations. Each comparative simulation was repeated 100 times and all results were used to create graphs and tables for AP performance evaluation.

For the comparative study here two algorithms were used - DE [Price 1999] and SOMA [Zelinka 2004]. A wide variety of optimisation algorithms, i.e. with different structures and their different abilities to locate the global extreme, were chosen to prove that AP can be regarded as an equivalent to GP (at least in the domain of used problems), and that it can be implemented using arbitrary evolutionary algorithms. As a conclusion, the following statements are presented:

1. **Results reached.** Based on results reported in Tables and Figures it can be stated that all simulations give satisfactory results and thus AP is capable of solving these classes of problems.
2. **Mutual comparison.** When comparing both algorithms, it is apparent that both algorithms give good results. Parameter settings for both algorithms were based on a heuristical approach and thus there is a possibility that better settings can be found.
3. **Universality.** AP was used to solve differential equations [Zelinka 2002 b], trigonometrically data fitting [Zelinka 2002

a], four polynomial problems from [Koza 1998] (Sextic, Quintic, Sinus Three, Sinus Four) by four EAs in [Zelinka and Oplatkova, 2003] and Boolean even - k - parity and k - symmetry functions synthesis [Zelinka and Oplatkova, 2004], [Zelinka, Oplatkova and Nolle 2004] and [Zelinka, Chen, Celikovsky, 2008]. Together with the results reported here, it can be stated that AP seems to be a universal method for symbolic regression by means of arbitrary EAs.

4. **Failure.** The synthesis of selected dynamical systems failed for system (13). One of possible explanations of this event is that the cost function or algorithms parameters should be defined in a different way.

Future research is one of the key activities in the frame of AP. According to all results obtained during the time it is planned that the main activities would be focused on:

- Expanding of this comparative study for genetic algorithms and simulated annealing, (and other evolutionary techniques).

ACKNOWLEDGEMENT

This work was supported by grant No. MSM 7088352101 of the Ministry of Education of the Czech Republic and by grants of the Grant Agency of the Czech Republic GACR 102/06/1132.

REFERENCES

Johnson Colin G., 2003, “Artificial immune systems programming for symbolic regression”, In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, Genetic Programming: 6th European Conference, LNCS 2610, p. 345-353

Koza J. R., M. A. Keane, M. J. Streeter, 2003, “Evolving Inventions”, Scientific American, February 2003, p. 40-47, ISSN 0036-8733

Koza J.R. 1998, “Genetic Programming II”, MIT Press, ISBN 0-262-11189-6, 1998

Koza J.R.,Bennet F.H., Andre D., Keane M., 1999, “Genetic Programming III”, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999

Lampinen Jouni, Zelinka, Ivan. “New Ideas in Optimization - Mechanical Engineering Design Optimization by Differential Evolution”, Volume 1. London : McGraw-Hill, 1999. 20 p. ISBN 007-709506-5.

O'Neill M. and Ryan C. 2002, “Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language”. Kluwer Academic Publishers, ISBN 1402074441

O'Sullivan John, Conor Ryan, 2002, “An Investigation into the Use of Different Search Strategies with Grammatical Evolution”, Proceedings of the 5th European Conference on Genetic Programming, p.268 - 277, 2002, Springer-Verlag London, UK, ISBN:3-540-43378-3

Price K. 1999, “An Introduction to Differential Evolution, in New Ideas in Optimization”, D. Corne, M. Dorigo and F. Glover, Eds., s. 79–108, McGraw-Hill, London, UK, 1999. ISBN 007-709506-5

Ryan C., Collins J.J., O'Neill 1998, M., “Grammatical Evolution: Evolving Programs for an Arbitrary Language”, Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming 1998

Zelinka I., 2002 a, “Analytic programming by Means of Soma Algorithm”, Mendel ’02, In: Proc. 8th International Conference on Soft Computing Mendel’02, Brno, Czech Republic, 2002, 93-101., ISBN 80-214-2135-5

Zelinka I., 2002 b, “Analytic programming by Means of Soma Algorithm”, ICICIS’02, First International Conference on Intelligent Computing and Information Systems, Egypt, Cairo, 2002, ISBN 977-237-172-3

Zelinka I., Oplatkova Z., Nolle L., 2005, “Analytic Programming – Symbolic Regression by means of arbitrary evolutionary algorithms”, IJSST, Vol 6, No 9

Zelinka I., Chen G., Celikovsky S., 2008, “Chaos Synthesis by Means of Evolutionary Algorithms”, International Journal of Bifurcation and Chaos, University of California, Berkeley USA, Vol 18, No 4, p. 911 - 942

Zelinka I., Oplatkova Z., 2003, “Analytic programming – Comparative Study”, CIRAS’03, The second International Conference on Computational Intelligence, Robotics, and Autonomous Systems, Singapore, 2003, ISSN 0219-6131

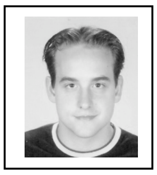
Zelinka I., Oplatkova Z., 2004, “Boolean Parity Function Synthesis by Means of Arbitrarry Evolutionary Algorithms - Comparative Study", In: 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004), Orlando, USA, in July 18-21, 2004

Zelinka I., Oplatkova Z., Nolle L., 2004, “Boolean Symmetry Function Synthesis by Means of Arbitrarry Evolutionary Algorithms - Comparative Study", In: 18th European Simulation Multiconference (ESM 2004), Magdeburg, Germany, June 13-16, 2004, ISBN 3-936150-35-4

Zelinka Ivan, 2004, “SOMA – Self Organizing Migrating Algorithm“,Chapter 7, 33 p. in: B.V. Babu, G. Onwubolu (eds), New Optimization Techniques in Engineering, Springer-Verlag, ISBN 3-540-20167X

AUTHOR BIOGRAPHIES

IVAN ZELINKA was born in Czech Republic, and went to the Technical University of Brno, where he studied technical cybernetics and obtained his degree in 1995. He obtained his Ph.D. degree in technical cybernetics in 2001 at Tomas Bata University in Zlin. He is now senior lecturer (artificial intelligence, theory of information) and head of the department. His e-mail address is: zelinka@fai.utb.cz and his Web-page can be found at http://www.fai.utb.cz/people/zelinka



ROMAN SENKERIK was born in the Czech Republic, and went to the Tomas Bata University in Zlin, where he studied Technical Cybernetics and obtained his degree in 2004. He is now a Ph.D. student and lecturer at the same university. Email address: senkerik@fai.utb.cz

ZUZANA OPLATKOVA was born in Czech Republic, and went to the Tomas Bata University in Zlin, where she studied technical cybernetics and obtained her degree in 2003, Ph.D. in 2008. She is now senior lecturer. Her e-mail address is : oplatkova@fai.utb.cz and her Web-page can be found at http://web.fai.utb.cz /?id=0_2_1_7&iid=3&lang=cs&type=0

Table 7 Results

System	Min N_{Eval}	Max N_{Eval}	∅ N_{Eval}	Min CV	Max CV	∅ CV
$\frac{1}{s^2+2s+1}$	31	3358	718,333	0	0,892707	0,0976925
$\frac{1}{(s+1)^3}$	56	2432	1110,81	0,494889	0,935361	0,768286
$\frac{1-0{,}5s}{s(s+1)}$	2	801	180,955	$3{,}1*10^{-14}$	0,685508	0,113692
$\frac{1}{(1+s)(1+0{,}5s)(1+0{,}5^2s)(1+0{,}5^3s)}$	38	221	99,3333	0,37409	0,512158	0,420114