

Symbolická identifikace

Symbolic Identification

Bc. Josef Vybíral

Diplomová práce
2005



Univerzita Tomáše Bati ve Zlíně
Fakulta technologická
Institut řízení procesů a aplikované informatiky

Vložit oficiální zadání diplomové/bakalářské práce

Prohlašuji, že jsem na celé diplomové práci pracoval samostatně a použitou literaturu jsem citoval.

Ve Zlíně, 02/09/2005

.....

Josef Vybíral

Abstrakt

Tato diplomová práce je shrnutím práce vykonané na implementaci symbolické identifikace pomocí analytického programování. Účelem této práce bylo urychlení výpočtů evolučního algoritmu použitím kompilovaného programovacího jazyka namísto interpretovaného jazyka typu Wolfram Mathematica®. Součástí práce je jak popis implementace analytického programování v jazyce C za použití evolučního algoritmu SOMA, tak i teoretická část obsahující shrnutí metod používaných pro symbolickou identifikaci a v obecné rovině pojednává o evolučních algoritmech. Závěr práce obsahuje vyhodnocení výsledků implementace analytického programování.

Abstract

This Master thesis is an overview of work done on programming of symbolical regression using analytic programming. Purpose of this thesis is to speed up evolutionary process of analytic programming by using of commonly used compiled programming language instead of special interpreted mathematical language (like Mathematica® language). Parts of this thesis are description of implementation of Analytic programming in C language using SOMA evolutionary algorithm, theoretical part covering methods used for symbolic identification. Some information about evolutionary algorithms are included too. Finally it includes evaluation of results of implementation of analytic programming.

ACKNOWLEDGEMENT

I would like to say thanks to Doc. Ing. Ivan Zelinka, PhD., for his patience and great support during my work on this thesis, for the information he gave me during my studies and finally for his lectures which brought me to the amazing world of artificial intelligence. Also I want to say thanks to Ing. Zuzana Oplatkova for her advices and consultations about analytic programming. After all I want to thank to all who supported me especially my friend Jana Tomasova from University of Ostrava for her support and help during debugging programs and writing this master thesis.

TABLE OF CONTENTS

1	Introduction to evolutionary algorithms.....	9
1.1	Basic information.....	9
1.2	Picked basic terms.....	10
1.2.1	Definition set of SOMA algorithm.....	10
1.2.2	Basic attributes of evolutionary algorithm.....	11
1.2.3	Population.....	11
1.2.4	Mutation and crossover.....	13
1.3	SOMA – Self-Organizing and Migrating Algorithm.....	14
1.3.1	Terminology and parameters.....	14
1.3.2	The population in SOMA.....	15
1.3.3	Mutation.....	15
1.3.4	Crossover.....	16
1.3.5	SOMA's main principle.....	17
1.3.6	SOMA strategies.....	18
1.4	Differential evolution.....	18
1.4.1	The terminology and parameters.....	18
1.4.2	The population.....	19
1.4.3	The Mutation.....	19
1.4.4	Crossover.....	19
1.4.5	The principle of DE.....	20
1.5	Addendum.....	21
2	The symbolic identification.....	22
2.1	Introduction.....	22
2.2	Genetic programming.....	22
2.3	Grammatical evolution.....	24
2.4	Analytic Programming.....	25
2.4.1	General Ideas.....	25
2.4.2	Versions of analytic programming.....	27
2.4.3	The mapping method.....	27
2.4.4	Crossover, Mutations and Other Evolutionary Operations.....	29
2.4.5	Reinforced Evolution.....	29

2.4.6 Security procedures.....	30
2.4.7 Similarities and Differences to GP and GE.....	31
2.4.8 Summary.....	32
3 The implementation.....	33
3.1 Implementation issues.....	33
3.1.1 Data files.....	33
3.1.2 General function set.....	34
3.1.3 Fitness function.....	34
3.1.4 Evaluation function.....	35
3.1.5 CostValue() function.....	37
3.1.6 Text construction functions.....	37
3.1.7 Function's length penalisation.....	39
3.1.8 Meta-evolution implementation.....	42
3.2 Known issues in the implementations.....	45
4 Example of implementation.....	46
4.1 Standard version.....	46
4.1.1 Changes in the main algorithm (der1b.c file).....	46
4.1.2 Cost value functions.....	47
4.1.3 Configuration and other changes.....	47
4.1.4 Addendum.....	47
4.2 Meta-evolution.....	48
4.2.1 Changes in the main file.....	48
4.2.2 Changes in costvalue() function.....	48
4.2.3 Changes in evaluation function (metaevol()).....	49
4.2.4 Changes of the costvalue2() function.....	49
4.3 Addendum.....	49
5 Implementations' results.....	50
5.1 Test functions and settings of evolutionary algorithms.....	50
5.2 Results of standard implementation.....	50
5.2.1 Quintic problem.....	50
5.2.2 Sextic problem.....	51
5.2.3 Sinus three problem.....	51
5.2.4 Sinus four problem.....	52
5.3 Summary.....	53

INTRODUCTION

The evolutionary algorithms were, during the time they exist, used for a wide area of optimization problems. This area grows more and more with researches done. New methods of symbolic identification are the new optimization problems that can be solved by an evolutionary algorithm too. The symbolic identification means that mathematical or programme functions are constructed on the basement of requested results. This problem has been solvable only by human's brain for a long time ago. But now, thanks to the computers power and the evolutionary algorithms, we have got the solution how to find the solutions.

During the time a few different ways how to use evolutionary algorithms for this purpose were develop. These methods are the Genetic programming, the Grammatical evolution or new method called the Analytic programming. The last one, the Analytic programming is a subject of interest of my master thesis. The Analytic programming has the big disadvantage in the meantime – it is programmed in interpreted programming language and because of this, the time to find any results takes too long. So the power increase of the computation is the main goal of my master thesis.

To achieve the goal, I'll realize algorithms of the analytic programming in classic(compiled) programming language using the evolutionary algorithm SOMA.

This master thesis is divided into several parts. In the first one, the basics of evolutionary algorithms and SOMA algorithm are described. The second chapter is shortly about existing methods of symbolic identification, their advantages and disadvantages. The third chapter is more detailed description of the Analytic programming and in the fifth part it describes the implementation of the analytic programming in C programming language. In the last chapter are tests and comparisons described with graphs of simulated data.

1 INTRODUCTION TO EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are a subspace of so called optimization algorithms. These algorithms are very powerful when used to find solutions of many technical problems. The optimization algorithms, so the evolutionary too, are used especially where analytic ways of solution fail.

The most of developed problems are optimisation problems. These problems can be for example an optimal setting of neural network's weights, optimal settings of automatic controller etc. In other words, every problem we can mathematically describe can be “optimisation problem” and this description is so called Cost Function. By optimisation of this cost function we can find its best fitting parameters which can be integer, real or complex values. There can be applied any penalizations for parameters or results during the evolution process too. The final result is, that we can solve any problems by more efficient way then by using analytic methods of optimization.

The typical feature of the evolutionary algorithms is usage of so called “*populations of possible solutions*” which are called individuals. These individuals interact during the evolution among themselves in cycles called *Generations* and target of whole evolutionary process is to find the best solution.

In this chapter some basic information about evolutionary algorithms is described and some terms used later in the thesis are also clarified. This chapter presents two representatives from the big set of evolutionary algorithm – SOMA and Differential evolution. These two were chosen because of their later use in testing programs. This presentation is tend to be a short informational overview, for closer information about these or any other EA's please, refer to the used literature on books e.g. [5] [14], [15].

1.1 Basic information

Optimization methods can be split into 4 categories(in accordance to method they use), enumerative methods – based on evaluation of all possible combinations of parameters of costvalue function. Deterministic methods – these are based only on the strict principles of mathematics. Stochastic methods – using circumstance to find the combination of parameters. And finally Mixed methods which are combination of previous. Really large set of these methods are evolutionary algorithms.

The basic of every evolutionary algorithm is cost value function's definition. This function's optimization(we look for its minimum or maximum) mostly leads to the right solution. So a definition of this function is one of the critical points in optimization with strong influence on quality of solutions.

1.2 Picked basic terms

The most of terms described in this chapter are related to the SOMA(Self Organizing and Migrating Algorithm) algorithm and Differential evolution because of their use in this thesis. But the most of terms can be applied to any evolutionary algorithm in general.

1.2.1 Definition set of SOMA algorithm

The *Definition set* term usually means suitable set of a number for which is the function defined. The definition set for evolutionary algorithm is set, which gives satisfactory results. Every optimization problem can be visualized as a geometric problem and target is to find a minimum or maximum on N-dimensional hyperspace. Hyperspace is usually defined by cost value function. These functions can have some pathological effects(in mathematical term). According to tests which were done we can say that from functions which:

1. have non-fractal type
2. are defined on real, integer, complex or discrete arguments
3. are unimodal or multimodal(have one or more extremes)
4. have various restrictions(for arguments or cost value's value)
5. have multiple objectives
6. are linear or non-linear
7. represents problems like “needle in a stack of hay”
8. belong to the NP problems' class

and for any type of function or their combinations can be valid that:

- a) interaction degree is hi/low
- b) function is separable – not separable
- c) number of variables is hi/low
- d) space of possible solutions is
 1. small – large
 2. finite – infinite
 3. continuous – discrete

was SOMA algorithm tested[16] on the most of the types of functions. Generally we can say that evolutionary algorithms can be used on “any” function which returns scalar value(except point no. 7).

1.2.2 Basic attributes of evolutionary algorithm

Evolutionary algorithms like SOMA and others have several common attributes(passed from [11]):

1. Simplicity. These algorithms are easy to program.
2. Variability of numbers used in algorithms. There can be mixed types(real, integer...) in the individual without any problem.
3. Numbers are in decimal values. These values don't need to be converted to binary values used in genetic algorithms for example.
4. Speed.
5. Independence of cross-over on the quality of parents. Quality based cross-over used in genetic algorithm can lead EA to the local extreme. SOMA generates sequences of offspring (point on hyperspace). This guarantees good search of the hyperspace.
6. Ability to find an extreme on function which are(in graphical way of meaning) flat and extreme is only a hole in this surface. (“Needle in a stack of hay” problem) Usually the probability that any evolutionary algorithm finds this kind of extreme is really low.
7. Ability to give multiple solutions. The result of the evolution is the best individual. But the result can be two or more best individuals too. Then they represent other possible solutions.
8. Efficiency in solving non-linear problems with boundaries (usually with some penalization function).

By other words, evolutionary algorithms are algorithms based on searching of extremes of functions afflicted by such a pathological attributes like high number of dimensions, multi-modality (more local extremes), parameters interaction etc...

1.2.3 Population

The work with population of individuals is the typical feature of any evolutionary algorithm. Population can be illustrated as matrix $D \times NP$ (see tab.I for example), where columns are individuals. Every individual represents possible solution of solved problem. In basics it is a set of parameters of cost value function and we search their values for the best combination. An additional information in every individual is its Cost Value.

The basic principle of evolutionary algorithms is to periodically do new and new populations and cross-over individuals in them. The cross-over acts in accordance to defined mathematical rules. These mathematical rules differ from one evolutionary algorithm to another and determine its type. Because the old population is replaced by the new one during the evolution and the population evolves, are these algorithms called evolutionary.

Table no. I example of population

Population	Individual 1	Individual 2	Individual 3	Individual 4	Individual 5	Individual 6
Cost Value	0,96991405	0,23794776	0,09628822	0,8579031	0,67769403	0,84439045
Parameter 1	0,03482728	0,25006514	0,10106705	0,79084894	0,13873141	0,69246614
Parameter 2	0,92125128	0,82017595	0,15929379	0,63566517	0,07897227	0,79798589
Parameter 3	0,23075334	0,43989923	0,4616472	0,04846819	0,21742956	0,58704542
Parameter 4	0,5843778	0,38566217	0,79953353	0,43016844	0,89554174	0,32675151
Parameter 5	0,0393167	0,34062729	0,51190694	0,64939382	0,90801454	0,8220979

The population has to be set on the beginning of each evolutionary process. The first definition of the population is done by a random number generator and is based on the prototype of an individual. This prototype is called *Specimen*. This specimen is then used for corrections of parameters of individual which run out of boundaries during the evolution. The structure of the specimen is specified by (1).

$$Specimen = \{ \{ type, \langle Lo, Hi \rangle \}, \{ type \langle Lo, Hi \rangle \}, \dots \} \quad (1)$$

Each parameter of individual is in the specimen defined by three parameters:

1. Type of the parameter's variable(integer, real, discrete, etc...).
2. Boundaries of the interval of values which can variable has.

Boundaries are very important in an individual's definition. The solution that is found on basics of bad interval can be physically unrealisable(negative weight e.g.) or will not be useful(aeroplane without wings...).

The second implication of boundaries is from the point of view of the evolution. There can happen that given optimisation problem is represented by an area where local extremes can gain greater values with greater distance from the beginning(for example Schwefel's function). This can cause that evolution can run into infinite if no stopping rules are defined. It is caused that the solution moves to a more and more deeper extremes of cost value function.

The population is then generated using (2) where x^{Hi} and x^{Lo} are interval boundaries from the specimen.

$$P^{(0)} = x_{i,j}^0 = rnd_{i,j}(x_i^{Hi} - x_i^{Lo}) + x_j^{Lo} \quad i=1, K, n_{pop}, j=1, K, n_{param} \quad (2)$$

This formula guarantees that values of individual will be from the interval we want. The population's initialisation is a random process. Thanks to this the individuals are spread randomly over the whole hyperspace. During the run of evolution, individuals are concentrating at a global extreme. The quality of the evolution can be discovered from the graph of History of evolution of the cost value. It is a sequence of cost function's values depending on the number of generation. An example can be figure no. 1.

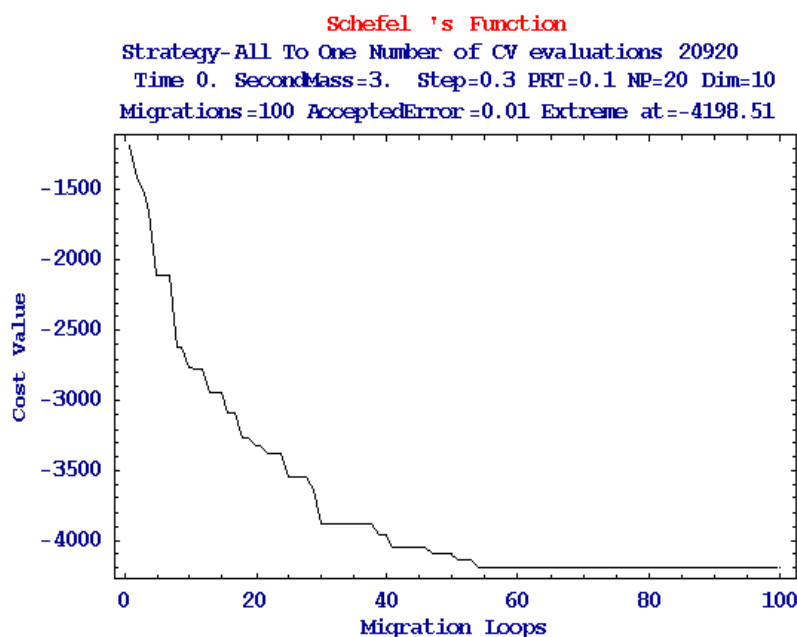


Fig. 1 - history of cost value

The progress of the population has to move to better values every time. In case of searching for the minimum, the progress has to aim to lower values and contra. If not, than there is broken an “elitism” principle in the evolutionary algorithm which works like an one-way filter to let only better individuals go the new population.

If this does not work, the algorithm degrades to random search which has practically nothing to do with evolutionary algorithms.

1.2.4 Mutation and crossover

The principle is the same as in Ch. Darwin's theory about The origin of species. There are some influences which affect the population during the evolutionary process. In the case of crossover is it a choice of parents and their attributes projected into their offspring. The algorithm of choice is different for different evolutionary algorithms. The number of parents don't have to be exactly two. Some algorithms like Differential evolution use four parents for example.

The mutation can be then regarded as the influence of the environment. The effect of the mutation is that evolution can find new and better individuals and improve the population. In the case of evolutionary algorithm is the mutation mostly realised by multiplying the parameter of the individual by predefined or randomly generated constant.

1.3 SOMA – Self-Organizing and Migrating Algorithm

SOMA is relatively new evolutionary algorithm developed by Doc. Ing. Ivan Zelinka, PhD. in 1999. It is not an ordinary evolutionary algorithm. It differs from the group of standard evolutionary algorithms in the evolution process. There are no new populations created during the evolution process. Algorithm works with original population all the time and changes it. More accurate label of this algorithm is Memetic algorithm(see [1]. The memetic algorithms' principles were defined as strategies of bevy running for its leader – competitive-cooperative strategy[6] to imitate behaviour of intelligent individuals who cooperate on solution of the problem. SOMA uses this principle and during the time of development, there were implemented different strategies.

The self organisation effect occurs thanks to the movement of individuals over the searched hyperspace. During this movement are created or destroyed groups of individuals. On the end of the evolution/migration is the most of individuals(or all of them) grouped in the extreme.

1.3.1 Terminology and parameters

As in other evolutionary algorithms is SOMA's run affected by parameters which can be sorted as control and termination parameters. Control parameters affect run of the algorithm, termination parameters stop the algorithm. The most of parameters have recommended intervals of use. These intervals/values has an influence on sensitivity of an algorithm and if they are set wrong, than the algorithm can become insensitive or freeze on one place. The list of controll and terminating parameters with their description follows.

PopSize – size of the population – number of individuals in the population. recommended interval of this value is <10, user defined>. The 10 is the lowest boundary. Lower number of individuals causes algorithm to slow down. Maximum value is limited only by the difficulty of the problem and the power of the computer. The recommended value could be $\max. NP=10 \cdot D$.

Dimension – number of parameters of individual/problem – The number depends on a problem and its needs.

PathLength $\in <1.1, 3>$ – This parameter determines in what distance from the leader will actual individual stop his movement. PathLength=1 means that actual individual will stop his movement on the position of the leader. PathLength=2 that he will stop in the same distance as from the start to the leader. Maximal recommended value PathLength=3 was found as adequate for any problem during simulations.

Step $\in <0.11, \text{PathLength}>$ – Step is the step with the individual search the hyperspace during his movement to the leader. Lower values slow down the algorithm but the hyperspace is more detailed searched.

PRT $\in <0, 1>$ - **Perturbation**. The perturbation vector is generated according to this parameter. This vector then affects the movement of the individual. There is higher probability that the algorithm will lock in a local extreme. Because of this is recommended value 0.1.

Migrations – number of **migration laps**(like generations). Their number is defined by user and tells to the algorithm the number of evolution cycles. This is termination parameter.

MinDiv – Minimal Diversity – defines minimal difference between the best and the worst individual in the population to stop the evolution. If the absolute difference between their CV is lower than this value, the evolution ends. The value depends on a user but the recommended value is low value.

1.3.2 The population in SOMA

The population is created in the same way as it is described in chapter no. 1.2.3.

1.3.3 Mutation

In SOMA, the mutation is replaced by the perturbation. The reason is simple, the movement of individual across the hyperspace is perturbed, not mutated. Perturbation depends on the perturbation vector which is generated according to the PRT parameter. For each individual's parameter is generated a random number from the interval $<0, 1>$. Then is expression (3) used and finally we got a vector of 0 and 1.

$$\text{if } rnd_j < PRT \text{ then } PRTVect_j = 1 \text{ else } 0 \quad j=1, K, n_{param} \quad (3)$$

Table no. II Perturbation vector example with 5 parameters and PRT=0.4

rnd_j	$PRTVect_j$
0.234	1
0.422	0
0.325	1
0.157	1
0.438	0

The perturbation vector is created before the individual starts to move across the hyperspace. The influence of the perturbation vector is on the movement. The result is that individual does not move in N -dimensional space, but only in $N-k$ dimensional space where k is number of zeros in perturbation vector. This helps to search the hyperspace and spread movement of individuals on their way to the leader. Thanks to this the evolution has better chance to find global extreme.

1.3.4 Crossover

The crossover is differently realized in the SOMA then in another evolutionary algorithms. SOMA does not create any new individuals as a combination of two or more parents. The crossover here means the move of the individual across the hyperspace and map of the path of the move. During this movement individual remembers the best found position and on the end of the movement moves to this position. From this position the individual continues in the next migration lap.

The behaviour of the individual during the migration is controlled by formula (4) or also can be controlled by (5).

$$\vec{r} = \vec{r}_0 + \vec{m} t PRTVec \quad \text{where } t \in \langle 0, Step, Mass \rangle \quad (4)$$

$$x_{i,j}^{ML+1} = x_{i,j}^{ML} + (x_{L,j}^{ML} - x_{i,j}^{ML}) t PRTVect_j \quad \text{where } t \in \langle 0, Step, Mass \rangle \quad (5)$$

The x_L parameter is a leader, an individual with the best value of the cost function. As described in formula (5), perturbation vector affects the individual's movement only if there are some zeros in it. If there is a zero in the perturbation vector, the affected parameter does not move. The effect of the perturbation is in better diversibility of the population and the algorithm is more robust.

1.3.5 SOMA's main principle

As I described before, SOMA is based on cooperative-competitive philosophy, That means that individuals compete and cooperate to find the best solution of the problem. This philosophy is taken from the real world and work in two phases. In competition phase, individuals search the hyperspace to find the best solution they can. In cooperation phase, they share the information about the best found solution and migrating to it. There are new and better positions found where the solution is better so the competition/cooperation continues.

This simple principle allows SOMA to find mostly global extremes on searched hyperspace. The description of the algorithm is, with references to previous chapters, following:

1. **Parameters definition** – In this step, user has to specify values of control and terminating parameters. Beside this parameters user has to specify definition of the specimen and cost value function. Cost value function would return scalar value representing quality of solution.
2. **Population initialisation** – In this step, the algorithm initialises the population using the specimen and random numbers generator. The initialisation uses formula (1).
3. **Migration lap** – The individual is evaluated on the beginning of the every migration lap. From the evaluated individuals, the leader is chosen as the one with the best CV. After this the migration starts and individuals migrate step by step to the leader. On every position is individual evaluated again and remembers the position if he finds better then he has saved. Migration to the leader is over when individual reach the position given by PathLength parameter. New positions are computed using formulas (4) or (5). For usage in algorithm is better to use formula (5).
When the migration is over, the individual returns to the best found position. After all individual's migration, they all have changed their position except the leader. Then the new leader is selected and the migration begins again.
4. **Evaluation of terminating conditions** – in this step the termination conditions are evaluated. If any of them is valid, migration is over and program shows the result it found. These conditions are the number of migration laps and Minimal Diversity value. If they are invalid, the evolution continues by step 3.

1.3.6 SOMA strategies

The modifications of SOMA are called “strategies”. The basic strategy is All to One and is described in previous chapter. All strategies are:

1. **All to One** – its principle was previously described. The individual moves to another one, the leader.
2. **All to One Rand** – In this strategy, all individuals move to one randomly chosen leader.
3. **All to All** – In strategy All to All leader does not exist. Individuals move in sequence to all other individuals from population during the migration. After moving to one, the individual move back to his starting position and starts to move to another one. After migrating all $\text{PopSize}-1$ laps he returns to the best found position.
4. **All to All Adaptive** – This modification works like the All to All modification but differs in one detail. In this strategy the individual does not change its starting position after the migration to all others but on the end of each migration. From this position he continues to move to another individuals.
5. **Meta evolution** – well this is not a real strategy as previous ones. But it is a way how can this or any other algorithm work too. There is a second evolutionary algorithm(SOMA e.g.) applied to search for optimised parameters of the cost value function constructed by main evolution.

1.4 Differential evolution

The Differential evolution was developed in 1995 by Ken Price and Rainer Storm. They had used it to find the solution of a problem with Chebysev's polynomials. Its principle is similar to genetic algorithms with which DE shares some characteristics. For example creating new offspring(using 4 parents, not only 2), run in generations etc...

1.4.1 The terminology and parameters

Activity and quality of Differential evolution is affected by its control parameters like in case of another evolutionary algorithms. Their sense is in the most cases the same as in any other evolutionary algorithm. In Differential evolution we use these:

- **PopSize – Size of the population.** Usually is this parameter from $\langle 2 \cdot D, 100 \cdot D \rangle$ but mostly are values set empirically according to experience. The recommended value is $10 \cdot D$. The only except in population creation is its minimal size. This shouldn't be less then 4. Bellow this value DE can't work.

- **CR – Crossover.** The interval of this parameter's values is $\langle 0, 1 \rangle$ and has an influence if or if not the mutation will affect the offspring. If $CR=0$, the mutation will not apply so the trial vector is then a copy of fourth parent. If $CR=1$, a trial vector consist only of a combination of three parents and then differential evolution works like random search algorithm. So the recommendation is to never set CR to these two values. The recommended interval is $\langle 0.3, 0.9 \rangle$.
- **F – Mutation.** Mutation constant is the last control parameter of the evolution. The value is from interval $\langle 0, 2 \rangle$.
- **GEN – number of generations.** This parameter tells the algorithm how many times he can do the evolution and search the hyperspace.
- **D – Dimension.** The number of arguments of cost value function. The final dimension of an individual is $D+1$. The last place is to store cost value.

1.4.2 The population

The Differential evolution works with population like any other evolutionary algorithm. For details see chapter 1.2.3 about populations and their initialization.

1.4.3 The Mutation

The Differential evolution differs from the evolution in the nature in the number of parents and that the mutation goes before the crossover. The number of parents is four. First three parents creates a noise vector which is their combination. In the concrete, the subtraction of first two is multiplied by the mutation constant and result is added to the third parent. This gives us a “noise vector”. Mathematical representation is formula (6).

$$v_{j,i}^{G+1} = x_{j,r3}^G + F \cdot (x_{j,r1}^G - x_{1,r2}^G) \quad (6)$$

1.4.4 Crossover

During the crossover is the noise vector(see previous chapter) transformed into a trial vector. For this is generated a random number for each parameter of the vector. This number is compared to the value of CR and if the value is greater or lower then CR so the value from noise vector or fourth parent is attached into the trial vector. The trial vector is then evaluated with the cost value function.

1.4.5 The principle of DE

The Differential evolution's purpose is to find extreme(global extreme) of cost value function during a number of generations of the evolution process. For this reason it cultivates a population of individuals. During every generation are done these steps:

1. **Parameters settings** – user can define parameters which control and terminate the evolution like F (mutation), CR (crossover), $PopSize$ (number of individuals, usually $PopSize=10*D$) and D (Dimension of the individual). The specimen has to be specified too for the algorithm of population initialisation. (For description of the specimen see chapter 1.2.3)
2. **Population initialisation** – Population is initialised as usual in evolutionary algorithms using the random number generator and specimen as a definition of an individual. The dimension of any individual is $D+1$ because of place to save the cost value function's value.
3. **The evolution** – During the evolution cycle is new population created. The reproduction process realizes as it is described in previous chapters about mutation and crossover. One parent is chosen in order, other three are chosen randomly. First two parents are subtracted, multiplied with “ F ” and then the third parent is added so we get a noise vector. Finally, according to a generated random number from $<0, 1>$ interval and value of CR , is noise vector converted to the trial vector using the fourth individual. If the generated number is greater than CR , the current parameter of noise vector is stored in the trial vector. If it is lower, stored value is from the fourth parent. This random value is generated for each parameter of the individual. When the trial vector is finished, then acts its evaluation. If the value of the cost value function is better then the CV of the first parent, the trial vector is copied into a new population. Else the first parent is copied into a new population as its offspring.
4. **Termination** – The whole evolution repeats until the number of generations is reached. During the evolution is the best individual from each migration saved to the history vector. On the end of the evolution is this history saved to show the progress of the evolutionary process.

The Differential evolution can used for meta-evolution like SOMA too.

1.5 Addendum

For details about SOMA or any other optimisation algorithm you can refer to the Internet or literature listed on the end of the thesis. The information about SOMA can be found on its homepage [21] too. For details about Differential evolution you can look at [20] or publications.

2 THE SYMBOLIC IDENTIFICATION

2.1 Introduction

The term *symbolic identification/regression* (SR) represents a process of fitting measured data by a suitable mathematical formula or a computer program. This process was only the domain of humans for a long time. But for the last decades it became the domain of computers too. There are two methods used for the purposes of the SR nowadays. The first one is *Genetic programming* (GP) ([2] - [4]) developed by J.R. Koza and the second one is *Grammatical evolution* [7], [11] developed by C. Ryan.

The idea of how to solve various problems using SR by the means of evolutionary algorithms (EA) was introduced by John Koza who used genetic algorithms (GA) for genetic programming. Genetic programming is basically a symbolic regression, which is using an evolutionary algorithm, instead of a human being, to develop the mathematical formula. The ability to solve very difficult problems was proved many times, and nowadays GP can be applied for example to synthesise highly sophisticated electronic circuits [4].

The second method is called *Grammatical evolution* and its author is C. Ryan. Grammatical evolution can be regarded as an unfoldment of GP because of some common principles, which are the same for both algorithms. The important difference and characteristic of GE is that it can be implemented in any arbitrary computer language. Contrary to the GP, which requires LISP programming language. In contrast to other evolutionary algorithms, GE was used only with a few search strategies, with a binary representation of the populations [8].

But there is a third, new and unique method of symbolic regression called *Analytic programming* (AP). This method was developed by Ivan Zelinka [12], [13], [17] and [18] and is the youngest from these three. Analytic programming is based on different principles compared to the GP and GE and its purpose is to be a universal method which can be used by any arbitrary evolutionary algorithm programmed in any computer language.

2.2 Genetic programming

The term *genetic programming* (GP) was for the first time used by the J.R. Koza [2]. Main principle of GP is based on genetic algorithm which is working with population of individuals and implemented in LISP programming language. Individuals used in GP are not binary strings as is usual for GA, but consist of LISP symbolic objects like sin, +, exp,

MyFunction, etc. These functions can be original LISP functions or user defined ones. Symbolic objects are usually divided into two classes: functions and terminals. Functions were just explained and terminals here represent a set of independent variables like x , y , and constants like π , 3.56, etc.

The principle of GP is usually demonstrated by using of so called trees (basically graphs with nodes and edges, Figure 2 and 3) representing individuals in LISP symbolic syntax. Individuals in the shape of a tree, or formula like $0.234z + x$ are called *programs*. Because GP is based on GA, then evolutionary steps (mutation, crossover, ...) in GP are the same as in the GA. As an example of GP can serve two artificial parents – trees at Figure 2 and 3 representing programs $0.234 * z + x - 0.789$ and $z * y * (y + 0.314 * z)$. When crossover is applied, then, for example, are exchanged subsets of trees beginning at nodes 2 (Figure 2 left) and 5 (Figure 2 right). Resulting offsprings are on Figure 3.

After the crossover and mutation is done, the fitness of the offspring is calculated to find out if the behaviour of just synthesized and evaluated individual-tree should be as much as

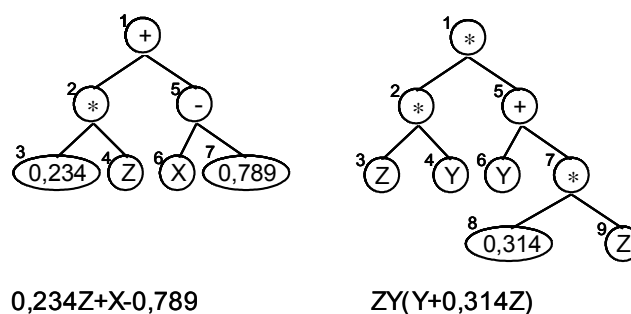


Fig. 2 - Parental program in genetic programming

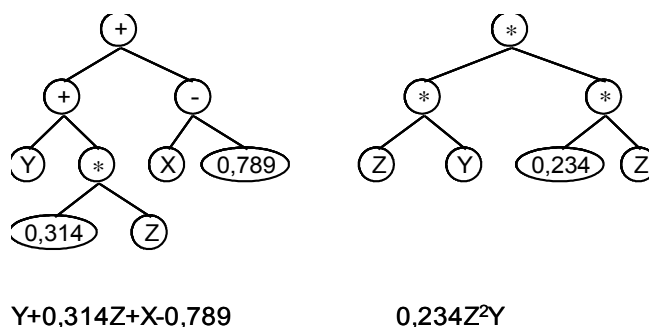


Fig. 3 - Offspring program in genetic programming

possible similar to the desired behaviour. Term *desired behaviour* can be understood like measured data set from some process (program should to fit them as well as possible) or like optimal commands for a robot's trajectory leading to the final position as best as possible. This is basically the same for GE and AP. For detailed description of genetic programming I recommend you to see [2] and [3].

2.3 Grammatical evolution

Grammatical evolution is a method based on GA, however thanks to a few important changes it has a few advantages comparing with GP. Main difference is individual coding. While GP is manipulating 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. The principle can be clarified by the following example.

Lets $T = \{+, -, *, /, X, Y\}$ is a set of operators and terminals and $F = \{expr, op, var\}$ so called *nonterminals*. In this case grammar used for final program synthesis is given by Table III. Rule used to transform individual into a program is based on (7). 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 IV.

$$\begin{aligned} unfolding &= codon \bmod rules \\ \text{where } rules &\text{ is number of rules for given nonterminal} \end{aligned} \quad (7)$$

Table no. III: 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 no. IV: Chromozome transformation

Chromozom e	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 program is following. Start begins with nonterminal object *expr*. Because integer value of Codon 1 (see table IV) is 40 then according to (III) is unfolding of $expr = op\ expr\ expr$ ($40 \bmod 2$, 2 rules for *expr*, i.e. (0) and (1)). Consequently Codon 2 is used for unfolding of *op* by $*$ ($162 \bmod 4$) which is terminal and thus unfolding for this part of program is closed. Then it continues in unfolding of remaining nonterminals (*expr expr*) till final program is fully closed by terminals. If program is closed before end of chromosome is reached, then remaining codons are ignored. Otherwise it continues again from beginning of chromosome. Final program based on just described example is in this case $X*Y$ (see figure 4). For fully detailed description of GE principles it is recommended to see [7].

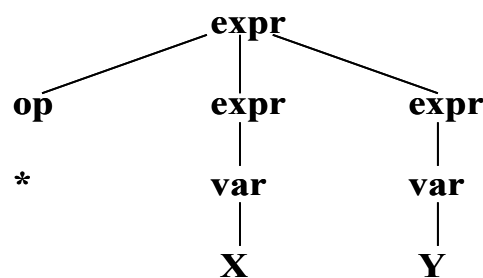


Fig. 4 - Tree structure of final program

2.4 Analytic Programming

2.4.1 General Ideas

The *Analytic programming* (AP) was inspired by the numerical methods in Hilbert functional spaces and by Genetic programming. The principles of AP are somewhere between these two philosophies. From GP was the idea of the evolutionary creation of

symbolic solutions and the general ideas of functional spaces and the building of resulting function are adopted from Hilbert spaces. Like both previous methods, the analytic programming is based on a set of functions, operators and so-called terminals, which are usually constants or independent variables, for example:

- functions: Sin, Tan, Tan, And, Or
- operators: +, -, *, /, dt,...
- terminals: 2.73, π , t,...

All these ‘mathematical’ objects create a set from which AP tries to synthesise an appropriate solution. The main principle of AP is based on discrete set handling (DSH), proposed in [15] (see Figure 5). Discrete set handling itself can be seen as an universal interface between EA and the problem to be solved symbolically.

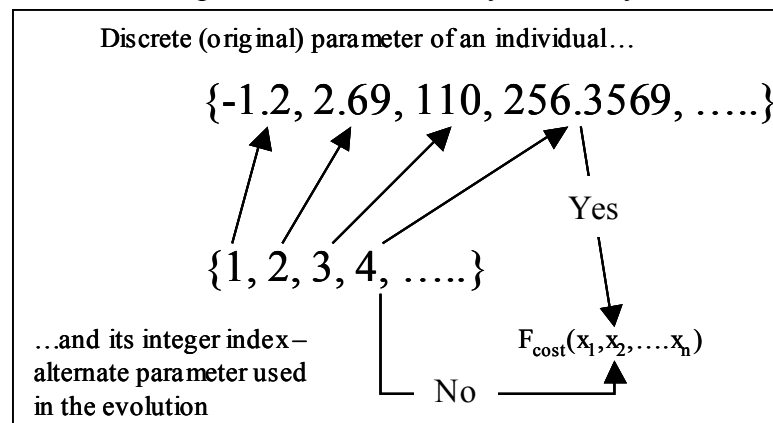


Fig. 5 - Discrete set handling.

The set of mathematical objects which DSH is using are functions, operators and so-called terminals (usually constants or independent variables). All these objects are mixed together to the set of functions. Because of the variability of the content of this set, it can be called “general functional set” – GFS. The structure of the GFS is presented by subsets of functions divided according to the number of their arguments (Fig. 6). The content of GFS depends only on a user and functions he wants to be used. In GFS can be various functions and terminals mixed together. For example GFS_{all} is a set of all functions, operators and terminals, $\text{GFS}_{3\text{arg}}$ is a subset containing functions with only three arguments, $\text{GFS}_{0\text{arg}}$ represents only terminals, etc.

In the evolutionary process, every individual consists of integer values that are indexes used as pointers into the GFS to concrete functions.

GFS_{all}	$= \{+, -, /, ^, \text{Sin}, \text{Cos}, \text{Tan}, t, x, \text{Mod}, \dots\}$
GFS_{3arg}	$= \{\text{User_Function}, \text{Beta}, \dots\}$
GFS_{2arg}	$= \{\text{Log}, \text{Mod}, \text{Gamma}, \dots\}$
GFS_{1arg}	$= \{\text{Sin}, \text{Cos}, \text{Tan}, \text{Abs}, \dots\}$
GFS_{0arg}	$= \{t, x, y, \pi, \dots\}$

Fig. 6 - Principle of the general functional set

2.4.2 Versions of analytic programming

The *Analytic programming* has three versions of the algorithm. All three versions use for program synthesis the same sets of functions, terminals, etc., as Koza uses in [2], [3]. The difference is in the method of searching the constants. First version (lets call it AP_{BASIC}) uses the same constants as described above. The second version (AP_{META}) has modified the technique of constant estimation. This technique uses only one constant. This constant is called K and is inserted into a formula at various places by the evolutionary process. When a program is synthesised, then all K 's are indexed so that $K_1, K_2, \dots, K_n$, appears in the formula, and then all K_n are estimated using a second evolutionary algorithm. Because EA (slave) “works under” EA (master) then this version is called AP with metaevolution - AP_{META} . This process is illustrated by formulas 8, 9 and 10.

$$\frac{x^2 + K}{\pi^K} \quad (8)$$

$$\frac{x^2 + K_1}{\pi^{K_2}} \quad (9)$$

$$\frac{x^2 + 3.56}{\pi^{-229}} \quad (10)$$

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 version uses a suitable method for non-linear fitting (AP_{NF}) to estimate constants. This method has shown the most promising performance when unknown constants are present. Results of some comparative simulations can be found in [12], [13], [17], [18], [19] or on the web [22].

2.4.3 The mapping method

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 are security procedures which do not allow to synthesise pathological programs. Discrete set handling proposed in [3], [15] is used to create an integer index, which is used in the evolutionary process to compute the cost function's value by method of integer handling. The method of DSH allows to handle arbitrary objects including non-numerical objects like linguistic (Hot, Cold,...) or logic terms (True, False) or other user defined functions. In the analytic programming, DSH is used to map an individual into GFS and together with security procedures it creates above mentioned mapping which transforms arbitrary individual into a program. As mentioned above, individuals in the population consist of integer parameters which point to the GFS at concrete functions.

An example of this mapping process can be illustrated by figure 7 and 8. Number 1 in the position of the first parameter mean that the operator “+” from GFS_{all} is used (the end of the individual is far enough). The operator “+” has to have at least two arguments, so next two index pointers 6 (sin from GFS) and 7 (cos from GFS) are chosen as arguments of this operator. 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 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 consist of subsets containing functions with the same number of arguments.

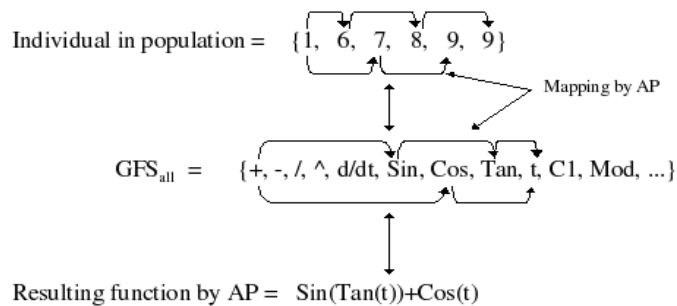


Fig. 7 - Main and simplified principle of AP

Existence of this nested structure is used in the special security subroutine which is measuring how far is the end of individual and according to this are selected objects from different subsets to avoid pathological function synthesis. Precisely if more arguments are desired than possible (the end of the individual is near) the function will be replaced by

other function with the same index pointer from subset with lower number of arguments. For example, it happens if the last argument for one argument function will not be terminal (zero-argument function) as demonstrated in Figure 8.

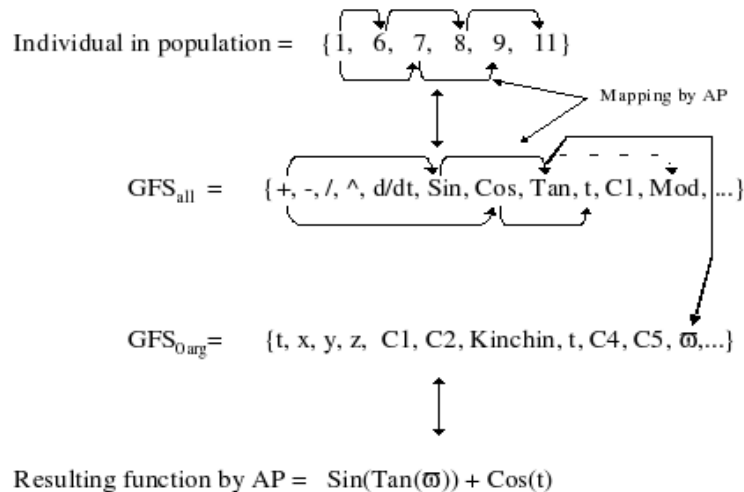


Fig. 8 - AP principle with security principles

According to main idea of AP a Mod function would be selected as an argument for Tan function (Fig. 8). When security subroutine measure how far is end of individual, then Mod is replace by ω from terminal subset.

GFS does not need to 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.

2.4.4 Crossover, Mutations and Other Evolutionary Operations

During evolution of a population, the AP does not affect the evolutionary process in any of its steps like crossover, mutation or other. From this point of view is AP only a layer between a evolutionary algorithm and cost function. This is next reason why is AP suitable for use in any evolutionary algorithm.

2.4.5 Reinforced Evolution

During evolution, less or more suitable individuals are synthesised. Some of these individuals can be used to reinforce evolution toward to better solution synthesis. Main idea of reinforcement is based on addition of just synthesised and partly successful program into

initial set of terminals. Reinforcement is based on user defined threshold used in decision which individual will be used for addition into initial set of terminals.

For example if threshold is set to 5, and fitness of all individuals - programs in the population is bigger than 5 then 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 initial GFS and is marked as terminal. Since this moment is evolution running on upgraded GFS containing partially successful program. Thanks to this fact is evolution able to synthesise final solutions much more faster than AP without reinforcement. This fact has been many thousand times verified by repeated simulations on different problems. When program is added into GFS, then threshold is also set to its fitness. If any individual with lower fitness is found then the old one is rewritten by new better one and threshold is reset to its value.

The reinforcement of individuals is quite similar to automatically defined functions (ADF) from genetic programming, however 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 to properly defined arguments of these ADF and critical situations checking (selfcalling,...). This is not problem of AP reinforcement, because 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.

2.4.6 Security procedures

Security procedures are in the AP as well as in GP, 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 integrated part of AP. However sometimes they have to be defined as a part of cost function, based on kind of situation (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 as a security procedure can be regarded here mapping from an integer individual to the program where is checked 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 that no pathological function will be generated. Another activities of SP are integrated part of cost function to satisfy items 2-4, etc.

2.4.7 Similarities and Differences to GP and GE

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

Similarities:

- Synthesised programs: 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 ([12], [13], [17], [18] and [19] similar to programs from GP and GE (see www.ft.utb.cz/people/zelinka/ap).
- Functional set: 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.

Differences:

- *Individual coding*: coding of an individual is different. Analytic programming uses an integer index instead of direct representation as in genetic programming. Grammatical evolution uses the binary representation of an individual, which is consequently converted into integers for mapping into programs by means of BNF [4].
- *Individual mapping*: AP uses discrete set handling, [15] while GP in its fundamental form uses direct representation in LISP [2] and GE uses Backus-Naur form.
- *Constant handling*: GP uses a randomly generated subset of numbers – constants [2], 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 non-linear fitting.
- *Security procedures*: 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 synthesised 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 synthesised before the end of the individual is reached (maximally when the end is reached).

2.4.8 Summary

Analytic programming seems to be a universal candidate for symbolic regression by means of different search strategies. It does not affect the evolutionary process so it can be used in any arbitrary evolutionary algorithm. Components of AP like DSH can be used as universal layer between the evolutionary algorithm and cost function. Finally, AP was tested on problems selected from many test and theory problems domain as well as from real life problems. They were like:

- Random synthesis of function from GFS, 1000 times repeated. This was to check if pathological function can be generated by AP. Individuals were randomly generated, evaluated by AP and checked for their internal structure. No pathological program was identified.
- $\sin(t)$ approximation, 100 times repeated. Here was AP used to synthesise program function $\sin(x)$ fitting.
- $\| \cos(t) \| + \sin(t)$ approximation, 100 times repeated, the same like in the previous example. Main aim was again fitting of dataset generated by a given formula.
- Solving of ordinary differential equations to find out if AP can find suitable resulting function.
- Boolean even and symmetry problems according to the [2] for comparative reasons.
- Simple neural network synthesis by means of AP.

The problems which are yet open or solved already:

- Optimal robot trajectory estimation (ESA conference).
- Controller synthesis – synthesization of controller for a selected class of systems.

3 THE IMPLEMENTATION

The implementation of analytic programming in C language can be split to two parts. The first one is the implementation of rules of analytic programming. The second is the implementation of evaluation of function represented by the individual. The second part is important because of absence of mechanism of symbolic calculations in C language. This chapter describes chosen solutions of these two problems and shows how they work with the memetic algorithm SOMA(All to One strategy) (see chapter 1.3 for description). The C language has been chosen because of its speed of program processing.

3.1 Implementation issues

The way of implementation differs between interpreted and compiled programming language. In the case of this implementation was important to choose the method of evaluation of function created by analytic programming at first. The first idea was to split an individual into blocks along two-argument functions. This way of implementation had problems with more complicated functions just because of the splitting along the two-arguments functions. This was the reason to choose different method. The chosen method is based on postfix languages and is described in this chapter.

3.1.1 Data files

After the definition of the way of individual's evaluation, the next issue was the definition of data file format. Finally chosen file structure is similar to the structure of the SOMA implementation's configuration file. It is a simple text file where each data is saved on separate row. The order and meanings of data is in the table no. V. This data vector can be automatically generated using a Matlab script I created for this purpose. Data are then stored to and loaded from `data.txt` file placed where other configuration files are placed.

Table no. V The structure of input data file `data.txt`

Length of data vector
Step of input variable
Initial value of input variable
Output data we have to fit ...

Data are loaded just after loading configuration of the evolutionary algorithm using newly defined function `CreateDataVec()`; This function loads data into the memory and reads values for initialization of variables.

3.1.2 General function set

The general function set consists of 3 types of functions in this implementation.

Terminals – constants or input variables. In the basic approach, the constants are 0-9 and PI.

In case of the meta-evolution, the set of terminals consists of input variables and constant K. All K constants are then fitted by the second evolutionary algorithm.

Functions with one parameter – these functions are function like algebraic or goniometric functions etc... (1p fcn for next)

Functions with two parameters – these functions are basic algebraic functions like +, -, * and /. (2p fcn for next)

The analytic programming can be used for the logical functions identification too. The function set than change to NOT, AND, OR, NAND, NOR and appropriate variables with generated truth table. All these functions are placed to their appropriate sets.

There are 26 functions defined in the GFS for standard implementation and 18 in meta-evolution version. These functions are:

GFS_{STD}: { {+, -, *, /, nth power}, {log₁₀, ln, exp, $\sqrt{x}$, x², x³, tan, cotg, sin, cos, abs}, {x, PI, 0 .. 9} }

GFS_{META}: { {+, -, *, /, nth power}, {log₁₀, ln, exp, $\sqrt{x}$, x², x³, tan, cotg, sin, cos, abs}, {x, PI, K} }

3.1.3 Fitness function

The function to compute cost value is the formula no. 11. This formula is evaluated for each data value of the input vector. Absolute value function is used instead of quadratic function because it gives better(bigger) results for comparison of cost values.

$$CValue = \sum_{i=0}^{DataLength} [|data_i - y_i|] \quad (11)$$

3.1.4 Evaluation function

The evaluation function combines algorithm of analytic programming with algorithm of final function evaluation into one recursive function. This function can be split into several parts according to the number of subsets of GFS. Each part can be then split into another parts which contain code for rules of analytic programming and algorithm of function evaluation.

The main principle of individual's evaluation is following. After reading the value from the individual, the function that matches to this ID in General Function Set is determined. Depending on kind of the function, required number of arguments is determined. This

number is than used in a condition which detects whether there is a required number of parameters left in the individual.

If there is not the required number of arguments for actual function available, the function is transformed into one of the set of function with less arguments. This update of function is realized by increasing the value by the number of functions in actual functions set. This transformation grants that the function is transformed into a terminal if there is no other argument.

If there is enough parameters available to fulfill function's requirements, then the identifier of function in the individual is increased and the recursive evaluation function is called from inside itself with a new argument. This procedure repeats until the parameter of the individual is terminal. In that case, the function returns the value of the terminal back to its superior function in the recursion. This function uses the returned value as an argument and returns next result, etc... until all functions are evaluated.

According to the principle described above, several variables and functions need to be defined to get this implementation working. The most important variables of this implementation are the numbers of functions in the separate subsets of GFS. These numbers are stored in variables named as `c0p`, `c1p` and `c2p` (two-argument functions are implemented as a maximum, but this implementation allows to rearrange to define functions using more arguments). These variables, together with variables for partial results (`v1` and `v2`), id of the actual parameter of the individual and variable "`rezerva`" are for a condition which presents the rule of analytic programming. The "`rezerva`" variable specifies the number of required but unused arguments of used functions. This condition is illustrated on the dumps no. 1 and 2.

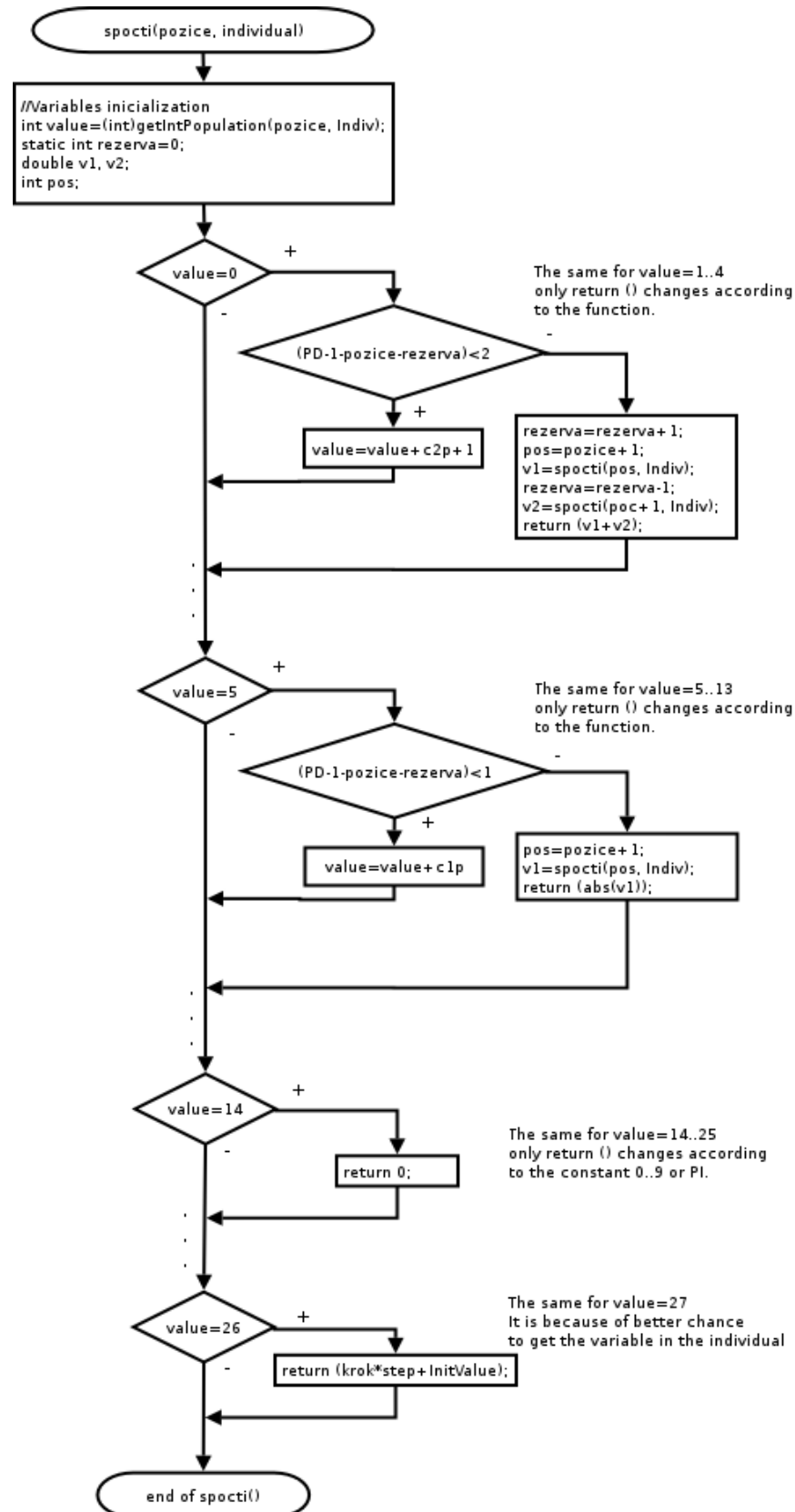


Fig. 9 - Flowchart of the algorithm of evaluation

dump no: 1 - code of the AP rule changing 2p to 1p function

```
if ((PD-1)-(pozice+rezerva)<2) { value=value+c2p+1; }
else { evaluation code }
```

dump no: 2 - code of the AP rule changing 1p function to terminal

```
if ((PD-1)-(pozice+rezerva)<1) { value=value+c1p; }
else { evaluation code }
```

The procedure mentioned above is only a segment of function called `spocti()`. This function consist of three switch statements which represent subsets of general function set. Every case from each switch statement contains one function from GFS and code required for its evaluation. At the beginning of each "switch" statement, there is a condition described above which determines the number of parameters available for usage as functions' arguments.

There are special conditions defined for some options of the switch statement. These conditions are defined to handle undefined mathematical operations such as division by zero etc... These conditions penalize the individual if it is valid. Whole function `spocti()` is listed on the dump no. 13.

3.1.5 CostValue() function

With `spocti()` function defined, the code of `costvalue()` function is short and simple. This code contains only definition of internal variables, one "for" cycle to handle all data from input file, count the cost value and function to save `CValue` back to the population.

dump no. 3 - full code of costvalue() function.

```
void CostValue(int Individual)
{
    double ResultStep;
    double CValue;
    CValue=0.0;
    krok=0;
    for(krok=0; krok<DataLength; krok++) // going thru the all data
    {
        ResultStep=spocti(0, Individual);
        CValue=CValue+(fabs(DataVec[krok]-ResultStep));
    } // end of FOR krok
    // save value of CValue into population
    setPopulation(PD-1, Individual, CValue);
}
```

3.1.6 Text construction functions

To create a text representation of a function presented by the individual is used, a slightly modified algorithm of its evaluation is used. That modification was to change the returned

value of the function from number to string. The newly created function is called `createText()`. The algorithm still uses evaluation part of the code from `spocti()` function to detect mathematical errors but has added functions to manipulate strings. These functions are mainly standard C functions like `strcat()` and `strcpy()` and a newly created function `addstring()`. This function uses the standard functions to easily manipulate operations and add function name to given string.

The `addstring()` function, listed on the dump no. 12 in attachment no. 1, is set to return text representation of the function using a syntax of Wolfram Mathematica package. I chose this syntax because of results' evaluation. It means that `addstring()` function closes the original string into two kinds of braces depending on the type of an argument and appends the name of the currently evaluated function to its beginning or end. Other brackets' syntax can be achieved by changing them in this function. Other functions names by changing strings in `createText()` function.

There are three input arguments of this function: the string we want to add, string we want it to be added to, and type of function and function placement argument. The Type argument can differ in four ways:

Type=0 : we want to add a function defined in Mathematica which is placed before its parameter(original string is an argument of this function). The parameter is closed into “[]” parentheses.

Type=1 : we want to add a mathematical function. The name of the function is to be placed after the parameter(usable for grouping items too). The parameter is closed into “()” parentheses.

Type=2 : we want to add a variable or constant as a parameter. In this case it adds no parentheses.

Type=3 : this option tells to `addstring()` that we want to add string before another string without adding any parentheses.

The principle of usage is similar to the evaluation algorithm. Parts of the text are recursively created (added using `addstring()` function) and combined together to make result. The `createText()` function is illustrated on the figure no. 11 and the function itself is listed on the dump no. 14 in attachment no. 1. The whole procedure acts in `CostValue2()` function which is called at the end of the evolution. After the creation of text representation of the function from individual, the final string has to be saved into the file with results so the `CostValue2()` function includes code to append the string to the file `result.txt`. The whole `CostValue2()` function is listed on the dump no. 4.

dump no. 4 - full code of costvalue() function.

```
void CostValue2(int Individual)
{
    double CValue, ResultStep;
    CValue=0.0;
    TextVec[0]='\0';
    TempTextVec[0]='\0';
    for(krok=0; krok<DataLength; krok++)
    {
        ResultStep=spocti(0,Individual);
        CValue=CValue+(fabs(DataVec[krok]-ResultStep));
    } // end of FOR krok
    printf("\n %s \n\n",createtext(0,Individual));
    setPopulation(PD-1,Individual,CValue);
    strcpy(ResFile,"result.txt"); // file to save to
    if ((out = fopen(ResFile, "a+")) == NULL)
    {
        printf("cannot open output file %s, creating new",ResFile);
        if ((out = fopen(ResFile, "wt")) == NULL)
            { printf("Cannot create output file %s !!!\n",ResFile);
              goto END;
            }
    }
    //return the best individual
    fprintf(out, "\n { %d, %lf,\n { %s },\n %s, 0 } \n", neval, CValue, TextVec,
    TextVec);
    fclose(out);
    END;;
}
```

3.1.7 Function's length penalisation

There is a string's length penalisation in the algorithm to prevent evolution to generate too long functions. The penalisation algorithm is simple. Value of 0.5 is added to the cost value of every function in the individual which participates on the result. But only once for the whole evaluation. The effect of this penalisation is in shorter output functions.

The penalisation is stored in the `str_penal` variable which algorithm adds to the `CValue` variable on the end of evaluation. Penalisation is a separate variable because of its later use.

I have implemented one more terminating condition to shorten the evolution process by other instruments than terminating parameters. Its purpose is to stop the evolution if the cost value of actual individual is near zero. The code from the dump no. 5 is therefore put after the code that inserts individual back to the population. The code requires yet another change of the main file to work. This change is in the dump no. 6 and looks for the best and the

worst individual. This code is put before the function which create the string representation of the individual because that function require its output. Such penalisation can rapidly shorten the evolution.

dump no. 5. - condition to terminate if CValue is near zero

```
if ((getPopulation(PD-1,i)-str_penal)<0.001) { jumped=1; goto SKIP;}
```

dump no. 6. - search the best and worst individual before text string creation if condition(5) is valid

SKIP:

```
if (jumped==1) {
    WorstIndividual();
    BestIndividual();
}
```

This code is included only in the standard implementation as an example of use.

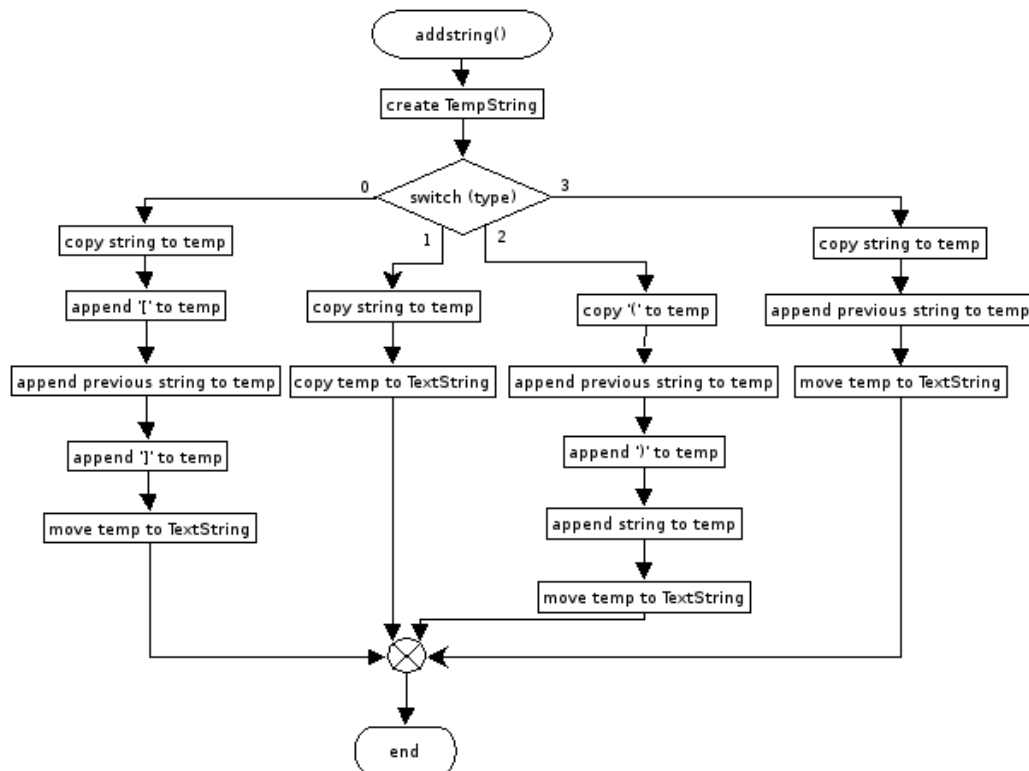


Fig. 10 - flowchart of the addstring() function

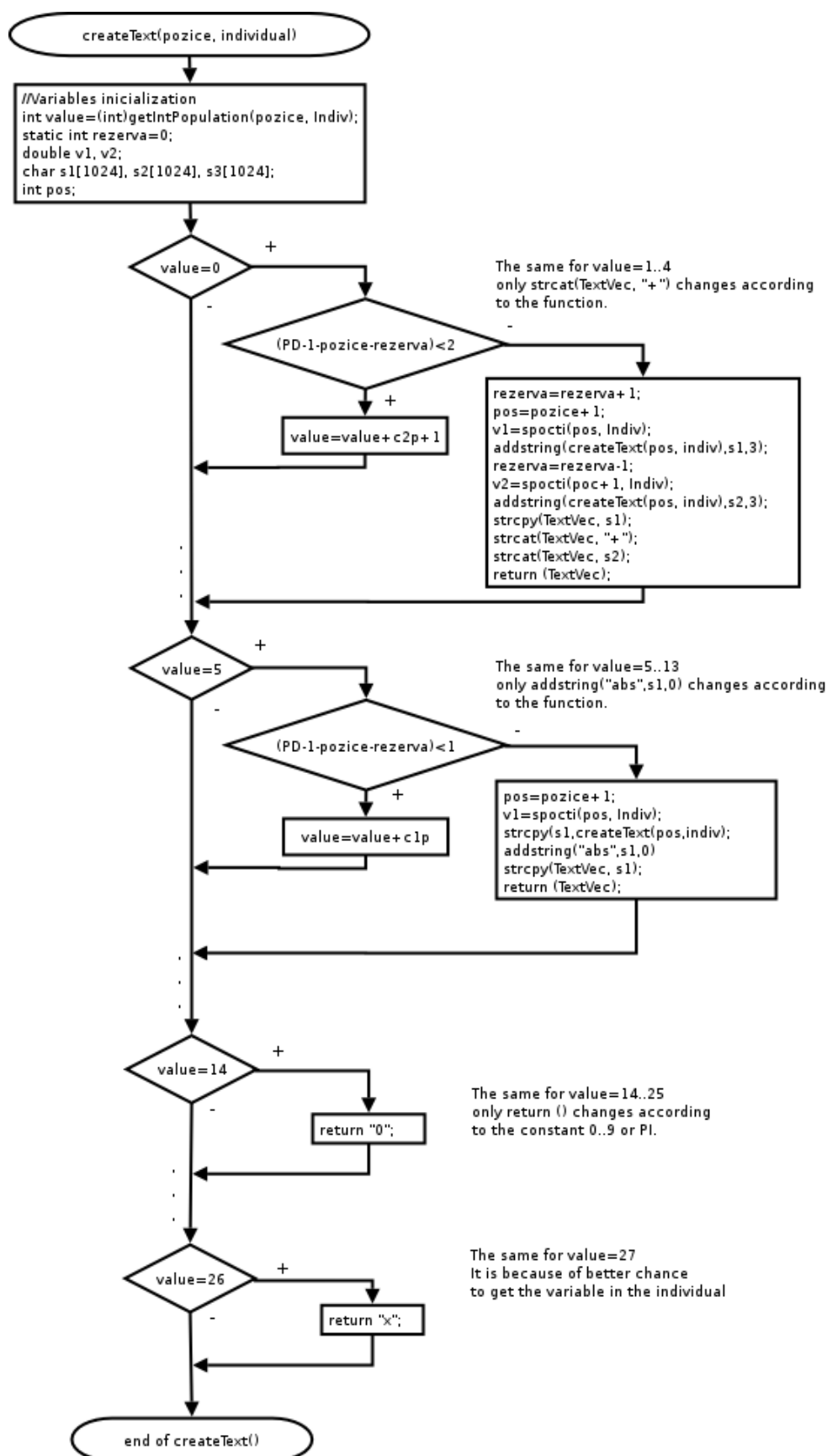


Fig. 11 - Flowchart of createText() function

3.1.8 Meta-evolution implementation

In the meta-evolution approach, another evolutionary algorithm is put into the `costvalue()` function instead of the evaluation algorithm. This evaluation algorithm is then in another function called from the `costvalue()` in this implementation. This is the main difference to the basic implementation and it requires the reimplementing of the second evolutionary algorithm with all of its functions and memory structures. I used the same EA for this purpose as the main evolutionary algorithm(SOMA).

The second difference to the basic implementation is the way of final construction of function's text interpretation. The main difference is in the way how values of constants are inducted into the text string. The first idea was that the implementation would use the second evolutionary algorithm to find constants again and then their values would be inducted into the string. The disadvantage of this solution is “circumstance”. In this way we don't know if the second evolutionary algorithm would be able to find the same constants which it found during the evolution. This was verified during some tests so this way is not the way this implementation uses.

This implementation uses the second idea. The best individual of the population is separately saved as an elite with all constants found(if there were some found). This elite vector updates in every generation/migration and finally we get the best one including constants found by the secondary evolution. Creation of a text interpretation of this individual is then simple and exact and uses previously defined functions with small modifications to fit the needs of meta-evolution.

Changes in the code/implementation

- Allocation of the memory to save the Elite. The code is simple and use is even simpler. The code on dumps 7. and 8. is placed in the `costvalue.c` file as all other functions required by this algorithm. These functions are called from the main algorithm and are two. `createElite(2*PD)`; after the population's allocation to allocate memory for elite (`2*PD` is the chosen size of the array to save individual and possible constants for it.) and `destroyElite()` at the end where memory is deallocated. The `*elite` is defined among other variables as a pointer.

dump no. 7 - allocate memory to save elite[]

```
int createElite (int pd)
{ elite = (double *) calloc (PPD, sizeof (double));
  if (elite == NULL)
    { puts ("Not enough memory to create elite vector !!!");
```

```

        exit (0); }
    return Pop ? 0 : 1;
}

```

dump no. 8 - deallocate memory for elite[]

```

int destroyElite ()
{ free (elite);
  elite = 0;
  return 0;
}

```

- **Counting the number of constants.** Before the evolution process there has to be put a function to determine the number of constants the EA is going to look for. This function is a modification of `spocti()` function and is supposed to **count the number of required constants** by the same way as `spocti()` evaluates the function. The new function is called `analyzuj()` and the difference against the `spocti()` function is in the part of constant's evaluation. On this place the function returns fictive constant and increases constants' counter variable `pocet_konstant`. The number of constants is important for the settings of second evolutionary algorithm.
- **Addition of a new evolutionary algorithm** to the `costvalue()` function. The second evolutionary algorithm replaces original content of `costvalue()` function. The original content (evaluation function) has to be moved to another function e.g. `Metaevol()` called from the second evolution as its fitness function. The second evolutionary algorithm has to be modified a bit too. There must be changed variables' and function's names because of possible name conflicts with variables and functions from the main EA.
- Addition of an algorithm that **saves better individual into the Elite**. This code is added twice. At the first time in the `costvalue()` function during the evolution and then on its end to save to elite the individual which does not contain any constants.

Dump no. 9 - algorithm to save the current better individual to the elite

```

if (CValue < elite[PD - 1])
{
    elite[PD - 1] = CValue;
    for (Mn = 0; Mn < PD - 1; Mn++)
    {
        elite[Mn] = getIntPopulation (Mn, Individual);
    }
    for (Mn = PD; Mn < Mpocet + PD; Mn++)
    {
        elite[Mn] = getIntPop (Mn - PD, Mk);
    }
}

```

The `elite[]` memory structure has to be globally defined variable to be accessed from any part of the program so the definition is on the beginning of `costvalue.c` file.

- **Change of the evaluation algorithm** to get constants from the population of the second EA. This change is done in order to remove fixed constants from the function set. The algorithm is replaced by one which loads constants using the
- algorithm on the dump no. 10.

Dump no. 10. - algorithm for loading the constants from the second evolution

```
case 14:    pos=pozice;
           Constant=getPop(pocet_konstant-1, MIndividual);
           pocet_konstant=pocet_konstant-1;
           return (Constant);
           break;
```

This code is modified in the function `analyze()` too. This is to count the number of constants(`pocet_konstant=pocet_konstant+1;`).

- **Changes of the `costvalue2()` function.** Because the best individual with its constants is saved in the `elite[]` array, the function `costvalue2()` and functions it uses had to be modified to read data directly from `elite[]` array instead of the original population. The function `costvalue2()` stays mostly unchanged, only the `analyze2()` function is added before the evaluation cycle, function `spocti()` is changed to the function `spocti2()` which uses `elite[]` as a source for evaluation and the same stands for the `createText2()` function.
- **Change of the `text construction function`** to properly add values of the constants to the string. The code is similar to the code used in `spocti()` function, only one row is added and some rows are changed. The added line is about the transformation of the value of constant from a number to a string(`sprintf(konst,"%lg",Constant);`). The variable `konst` is then returned in `return(konst);` statement. Another modification of this function is to read data from `elite[]` vector, which was mentioned above, instead of the original data from population.

These are essentially all the changes which are done in the meta-evolution. Of course all needed variables and functions have to be defined. In this case, these variables are `*elite` (the pointer to the array where elite is saved), and variables required by the second evolutionary algorithm. These changes are described in sample implementation of the meta evolution in next chapters.

3.2 Known issues in the implementations

- Because of some issues with `analyzuj()` function, which determines the number of constants required by function, the implementation of meta-evolution has not been tested yet and is quite unfinished.

4 EXAMPLE OF IMPLEMENTATION

In this chapter I would like to show you how can use my work in another evolutionary algorithm. For this example I chose an existing implementation of the Differential evolution algorithm. This algorithm is realized in C language and uses similar functions as SOMA's implementation. Anyone can download it from [21], section Source codes. Don't forget in any other implementation using other functions and variables, the concept of the cost value function I made has to be rewritten to use them.

4.1 Standard version

The implementation of the standard version of analytic programming into another evolutionary algorithm is easy. Only a few changes have to be done in the original algorithm's file, all other changes acts in functions' and configuration files. The only files that need to be edited in this case are `der1b.c`, `costvalue.c` and configuration file.

4.1.1 Changes in the main algorithm (`der1b.c` file)

The main algorithm of evolution needs no modification. But we have to add functions for input data loading and creation of needed memory structures at the beginning. The code is in the dump no. 16 in attachment no. 1.

I've put this code right after the code that reads the file and initializes parameters of the evolutionary algorithm. The second and last required change of the algorithm with the evolution is before its end. This one is to call `costvalue2()` function which creates the string representation of the best individual. The code is then:

```
CostValue2(best); // before the end of der1b.c
```

where 'best' is the index of the best individual in the population. For proper deallocation of memory we have to add functions to deallocate memory on the end of the main file.

```
destroyDataVec();
destroyFinalVec();
```

String length's penalisation

To add string length's penalisation you only have to add the code from dump no. 5. right after the replacement of the old individual by the new one. In the `der1b.c` file you can see it on the row 162 of the sources. The support code from the dump no. 6 have place just before call of the `costvalue2()` function on the end of the main algorithm. (row 185 in `der1b.c` file).

4.1.2 Cost value functions

Classic `costvalue()` function in the `costvalue.c` is in this case replaced by the prepared one from my `costvalue().c` file. The same happens with `costvalue2()` function which has not been used before. This function is in `costvalue.c` file too.

As I said before, `costvalue2()` function creates the final text representation of the result found by EA. This function has to be placed to the end of the main file(`der1b.c`) before memory deallocation functions.

4.1.3 Configuration and other changes

The last two things you have to do to get it fully working is to put any `data.txt` file in the directory where the executable is and change the configuration of evolutionary algorithm. The configuration change is to change specimen's specification(in the file `dejo.txt`). There are 27 functions defined which the algorithm can recognize(including constants and input variable) so evolution have to look for solutions only in the interval from 1 to 27 and as integer values. All other configuration parameters keep unchanged if user doesn't need to change them.

The implementation is complete now and you can try to compile the program. Don't forget to create `data.txt` file with data you want to analyse.

The final step is to compile source codes into a binary executable file. Type `make` command where your sources are located and wait for compilation to finish. This kind of compilation is usual in Linux/Unix environment with `gcc/g++` as a compiler. For other platforms and other compilers you have to adapt files like `makefile` to compile with your compiler.

4.1.4 Addendum

Of course this was an example, if you use another implementation of another evolutionary algorithm, you probably would have to do more then just copying prepared files and code into your programs. For this case you have to consult previous chapters about basic principles used to implement analytic programming in C language. These principles are described in details with flowcharts included so implementation in any other programming language would be easy.

4.2 Meta-evolution

The difference from the simple implementation of standard version is, that in meta evolution, we have to redefine some functions and memory allocations for a second evolutionary algorithm. Because of this the amount of added functions and variables further increases. Some changes described in the chapter about meta-evolution are required too.

4.2.1 Changes in the main file

If you implement meta-evolution using examples from this thesis, you only have to change the main algorithm's file adding two rows into it. These rows are concerning the creation and destruction of elite memory allocations so you have to add first command somewhere after functions creating population and add the second one to the end of the main file. function callings are:

```
createElite(2*PD);
destroyElite();
```

(The $2*PD$ is the estimated size of the array to save individual and possible constants for it.) Of course the changes to create and destroy data memory structure as described in the chapter 4.1.1 you have to do too.

4.2.2 Changes in costvalue() function

There is a one big change in the `costvalue()` function and several minor changes. The biggest change is the move of the evaluation code to the `metaevol()` function from `costvalue()` and insertion of the secondary evolutionary algorithm there.

I have reused the main algorithm to do this secondary evolution. All the code is changed, including functions to create and destroy population, specimen, read/save from/to the population etc... Of course all required variables were redefined as local for this function. The identifiers of functions were shorten(e.g. `getIntPopulation()` to `getIntPop()` etc...) and "M" is now the prefix of variables used by evolutionary algorithm. Also the code to save the result of evolution to the `elite[]` vector(dump no. 9) has to be placed at the end of second evolutionary algorithm.

The code that enumerates the number of constants to evolve needs to be inserted too. This code is on the dump no. 11.

Dump no. 11 - Code to initialize and enumerate the number of constants

```
pocet_konstanty=konstanty;
analyzuj(0, Individual);
Mpocet=konstanty;
```

```
pocet_konstant=konstanty;  
pocet_konstant_zaloha=pocet_konstant;
```

4.2.3 Changes in evaluation function (`metaevol()`)

The code of evaluation changed mainly in the code that is reading values of constants from the memory. The code change is listed on dump no. 10. All other code is the same as in the original evaluation algorithm.

The second change of the evaluation algorithm is the code at the end of this function. This code provides a condition whether to save the cost value from the evaluation directly to the population or return it back to the second evolution. This code is listed on dump no. 17.

4.2.4 Changes of the `costvalue2()` function

Because of use of the `elite[]` vector, functions used to evaluate the function and to create the text representation of the result had to be changed to use data from this vector. This means to change each occurrence of the `getIntPopulation(individual,parameter)` function to `elite [parameter]`. Functions reading constants from the population or transforming them into string have to be changed here too. The parameter for reading constants changes to `PD+pocet_konstant`.

After all there changes you can type `make` as in previous case and wait for a program to compile to an executable binary. This kind of compilation is usual in Linux/Unix environment with `gcc/g++` as a compiler. For other platforms and other compilers you have to adapt files like `makefile` to compile with your compiler.

4.3 Addendum

All changes I wrote about in this and previous chapter are completed into the relevant `costvalue.c` files. You can find these files on the CD-ROM included to this thesis. They are in the form of individual implementations and in the form of pure `costvalue.c` files in sources directory and its subdirectories. Of course binary executables for Windows and Fedora Core 3 are there too but compilation from sources is highly recommended due to performance of the application on specific system.

The meta-evolution implementation in the state of the development version is included on the CD-ROM too.

5 IMPLEMENTATIONS' RESULTS

5.1 Test functions and settings of evolutionary algorithms

The basic implementation of analytic programming was tested using several functions. These functions were chosen according to comparative studies at [22]:

- Sinus four problem
- Sinus three problem
- Sextic problem
- Quintic problem

All results of simulations of analytic programming are listed in the attachment no. 2 in the form of graphics charts. Simulations were done using SOMA and Differential evolution for basic implementation of analytic programming.

Settings for all used algorithms were:

SOMA A2O: PopSize=200, Dim=30, Migrations (depends on problem), Path Length=4, Step=0.7, PRT=0.2;

SOMA A2A: PopSize=200, Dim=30, Migrations (depends on problem), Path Length=4, Step=0.7, PRT=0.2;

DE: PopSize=300, Dim=30, Generations=3500, CR=0.6, F=0.7;

Source data for simulations were generated on interval $x \in \langle -1; 1 \rangle$ for Sextic and Quintic problems and on the interval $x \in \langle -\pi; \pi \rangle$ for Sinus Four and Sinus Three problems.

The configuration of computers where simulations ran are: 2GHz Intel Celeron CPU, 512MB RAM, platform Fedora Core 3 GNU/Linux; and Intel Centrino 1.6GHz, 512MB RAM, platform MS Windows XP Professional.

5.2 Results of standard implementation

5.2.1 Quintic problem

Quintic problem ($y = x^5 - (2 \cdot x^3) + x$) was easily solved by all three algorithms using standard implementation. The SOMA algorithm was more successful than Differential evolution according to the comparison on Fig. 12. One of the result functions found by evolution that fit the quintic problem on selected interval is (12).

$$y = x \cdot (\cos(x))^{(5 \cdot \cot(\cos(x)))} \quad (12)$$

Graphs of simulations results are on figures 16, 18 and 20 in attachment no. 2. There were 20 simulations with each evolutionary algorithm and all were successful.

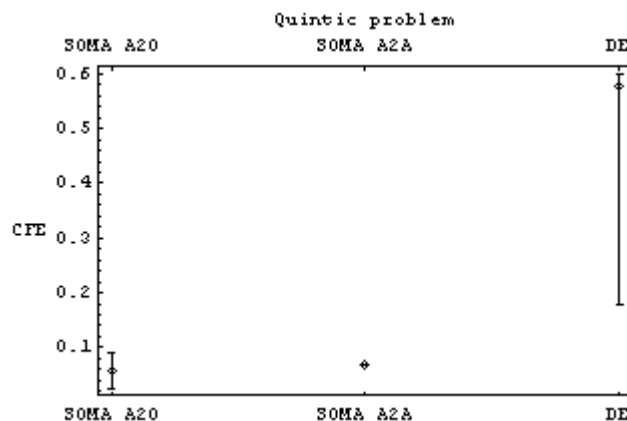


Fig. 12 - Variability of CV of SOMA and DE for Quintic problem

5.2.2 Sextic problem

The Sextic problem ($y = x^6 - 2 \cdot x^4 + x^2$) was also easy to solve for all algorithms. The best solution was provided by SOMA algorithm using All to all strategy (according to comparison on the figure 13) and Differential evolution provided the worst (from these 3 algorithms) but still satisfactory results. Charts on figures 22, 24 and 26 in attachment no. 2 show these results. From twenty of executed simulations were all twenty successful with all tested evolutionary algorithms.

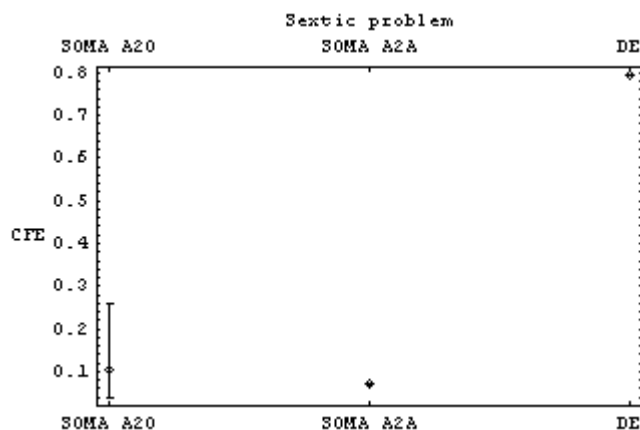


Fig. 13 - Variability of CV of SOMA and DE for Sextic problem

5.2.3 Sinus three problem

The Sinus three problem ($\sin(x) + \sin(2 \cdot x) + \sin(3 \cdot x)$) was more difficult to solve than previous two problems. Even if the fig. 14 shows higher values of CV for individual algorithms, the results as shown on figures 28, 30 and 32 say that selected found solutions were mostly successful. The best result was from SOMA with All to One strategy. The total number of executed simulations were 20 with each of evolutionary algorithms. Results were

following: SOMA A2O 7 successful functions, SOMA A2A - 3 successful ones and Differential evolution - 5 more or less successful functions. The sample function constructed by AP is (13).

$$y = 4 \cdot \sin(2 \cdot x) \cdot \left(\cos\left(\frac{x}{\sqrt{2}}\right) \cdot \sin(7) \right)^{(\cos(\sin(\cos(x))))} \quad (13)$$

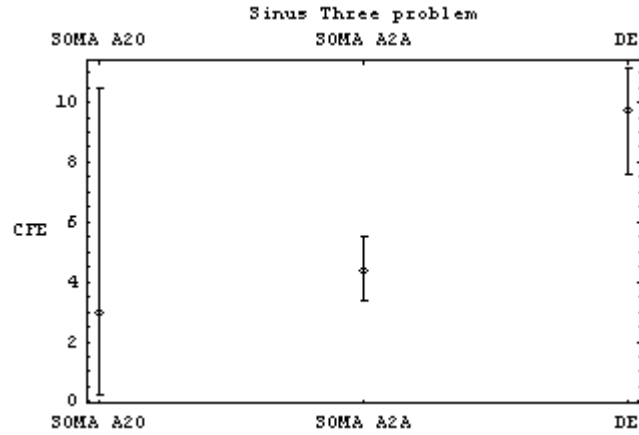


Fig. 14 - Variability of cost value of SOMA and DE for Sinus three problem

5.2.4 Sinus four problem

The sinus four problem ($\sin(x) + \sin(2x) + \sin(3x) + \sin(4x)$) was the most difficult function to be solved by the implementation of Analytic programming and three evolutionary algorithms. According to the comparison of variability of cost values (figure 15), the best results were given by SOMA using All to all strategy but no result was absolutely perfect.

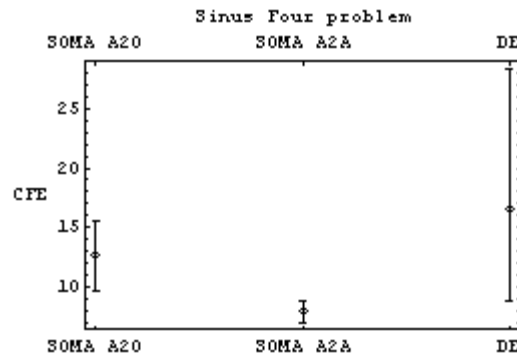


Fig. 15 - Variability of cost values of SOMA and DE for Sinus four problem

Formula (14) illustrates one of found solutions.

$$y = \frac{(2 \cdot \cos(\sin(\sin(\sin(6)) \cdot x)) \cdot |\sin(x \cdot \sqrt{5})|)^2}{x} \quad (14)$$

Charts of results of Sinus four simulations can be found in attachment 2 as figures 34, 36 and 38. There was a really big number of simulations required to get some good results for sinus four problem. The number of simulations was approximately 20 as in previous experiments. Each of the evolutionary algorithms found two more successful functions which are plot in charts mentioned above.

5.3 Summary

According to the results of standard implementation's simulations I can tell that simple functions are recognised with high accuracy. This accuracy of recognition lowers with complexity of the function. But according to the results of simulations, selected functions were fitted more or less successfully on selected intervals of input variable.

CONCLUSION

Analytic programming is fair method of symbolic identification. Its main advantages are simplicity and simple implementation. The implementations of the analytic programming described in this thesis are on the good way to speed up the evolutionary process and make analytic programming, especially in meta version, less time consuming. But they still need to be improved and debugged to solve more and more problems.

In the theoretical part, two evolutionary algorithms with their function and parameters were described. These chapters are important for understanding principles of evolutionary algorithms I had used for implementations. The main part of the thesis is description of issues of the analytic programming's implementation, In this part, the theoretical and practical ideas for implementations I used were described. This description includes source code fragments and flowcharts to help implementation of AP in another programming language than the one which was used.

Besides the algorithms of AP principles' implementation and function's evaluations were also developed functions for string handling. These functions can be reused if the target programming language does not have advanced functions for strings' operations.

Using the theoretical basics and developed algorithms were AP_{BASIC} and AP_{META} implemented to algorithms SOMA and Differential evolution. With successful implementation of AP_{BASIC} were then fulfilled experiments. Results of experiments were processed into graphical charts and described in experimental part of this thesis.

As the experimental part shown, the implementation of standard approach to Analytic programming was able to find solutions for all selected functions. Functions like Sextic and Quintic problem have been solved better than functions Sinus Three and Sinus Four. This could be caused by complexity of functions and the amount of input data which the algorithm can use. From selected evolutionary algorithms was SOMA(both used strategies) more successful than Differential evolution during simulations.

Besides the standard implementation of analytic programming there was a version of meta-evolutionary AP developed too. The final version of this implementation is not far away. The codebase is mostly complete and only a few issues have to be solved, the recursive method of number of constants enumeration for example. After debugging process the meta-evolutionary implementation will be able to work properly and show its power.

Finally, the code of all implementations uses C language so it can be compiled and tested on any platform where a C compiler is present. Standard compilation was tested on

development machine with Fedora Core 3 and on testing machine running Windows XP Professional, both compilations under GCC(G++) compiler were successful. These source codes and also binary executables are on the CD-ROM attached to this thesis.

The implementation in C language proved that the Analytic programming can be programmed in compiled programming language too. Thanks to that, the AP can be used in arbitrary systems with suitable programming language for the task of systems' identification or other tasks of automatic control.

BIBLIOGRAPHY

- [1] Dawkins R. *The Selfish Gene*. Oxford Univ. Press(Trade).
- [2] Koza J.R., Genetic Programming, 1998, MIT Press, ISBN 0-262-11189-6.
- [3] Koza J.R., Bennet F.H., Andre D., Keane M., Genetic Programming III, 1999, Morgan Kaufmann pub., ISBN 1-55860-543-6.
- [4] Koza J. R., M. A. Keane, M. J. Streeter, Evolving Inventions, 2003, Scientific American, p. 40-47, ISSN 0036-8733.
- [5] Kvasnička V., Pospíchal J., Tiňo P., Evolučné algoritmy, 2000, STU Bratislava, ISBN 85-246-2000.
- [6] Lampinen J., Zelinka, I. New Ideas in Optimization - Mechanical Engineering Design Optimization by Differential Evolution. Volume 1, 1999. London : McGraw-Hill, 20 p. ISBN 007-709506-5.
- [7] O'Neill M. and Ryan C., Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language, 2002. Kluwer Academic Publishers, ISBN 1402074441.
- [8] O'Sullivan John, Conor Ryan, An Investigation into the Use of Different Search Strategies with Grammatical Evolution Proceedings of the 5th European Conference on Genetic Programming, 2002, p.268 - 277, Springer-Verlag London, UK, ISBN:3-540-43378-3.
- [9] Price K., An Introduction to Differential Evolution, in New Ideas in Optimization, D. Corne, M. Dorigo and F. Glover, Eds., 1999, p. 79–108, McGraw-Hill, London, UK. ISBN 007-709506-5.
- [10] Ryan C., Collins J.J., O'Neill M. Grammatical Evolution: Evolving Programs for an Arbitrary Language. Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming 1998, 1998.
- [11] Zelinka, Ivan, Aplikovaná informatika. 1. vyd, Zlín: Vysoké učení technické v Brně, Fakulta technologická ve Zlíně, 1999, p. 183, ISBN 80-214-1423-5.
- [12] Zelinka I., Analytic programming by Means of Soma Algorithm. Mendel '02, In: Proc. 8th International Conference on Soft Computing Mendel'02, Brno, Czech Republic, 2002, p. 93-101., ISBN 80-214-2135-5.
- [13] Zelinka I., 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.

- [14] Zelinka, Ivan, Umělá inteligence v problémech globální optimalizace. 1. vyd., Praha: BEN, 2002, 232 s. ISBN: 80-7300-069-5.
- [15] 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
- [16] Zelinka Ivan, Lampinen Jouni, SOMA - Self-Organizing Migrating Algorithm, Mendel 2000, 6th International Conference on Soft Computing, Brno, Czech Republic, ISBN 80-214-1609-2.
- [17] 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
- [18] Zelinka I., Oplatkova Z., 2004, Boolean Parity Function Synthesis by Means of Arbitrary Evolutionary Algorithms - Comparative Study", In: 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004), Orlando, USA, in July 18-21, 2004, in print
- [19] Zelinka I., Oplatkova Z., Nolle L., 2004, Boolean Symmetry Function Synthesis by Means of Arbitrary Evolutionary Algorithms - Comparative Study", In: 18th European Simulation Multiconference (ESM 2004), Magdeburg, Germany, June 13-16, 2004, ISBN 3-936150-35-4
- [20] Storn R. (1996). *Differential Evolution* WWW: <http://www.icsi.berkeley.edu/~storn/code.html>
- [21] Zelinka, Ivan, SOMA, DE, Softcomputing, <http://www.ft.utb.cz/people/zelinka/soma>
- [22] Zelinka, Ivan, Analytic programming, <http://www.ft.utb.cz/people/zelinka/ap>

LIST OF USED SYMBOLS AND SHORTCUTS

0p	zero arguments function – terminal
1p	one argument/parameter function
2p	two arguments/parameters function
AP	Analytic programming
AP _{BASIC}	Basic implementation of AP
AP _{META}	Meta-evolutionary implementation of AP
AP _{NF}	Implementation of AP using non-linear fitting
CR	Crossover
D	Dimension of the individual
DE	Differential Evolution
DSH	Discreet set handling
EA	Evolutionary algorithm
F	Mutation in Differential evolution
GE	Grammatical evolution
GEN	Generations
GFS	Global function set
GP	Genetic programming
MinDiv	Minimal diversity
ML	Migration lap
PathLength	Length of the path of the individual in the SOMA
PopSize	Size of the population
PRT	Perturbation
PRTVec	Perturbation Vector
SOMA	Self Organizing and Migrating Algorithm

IMAGES LIST

Fig. 1 - history of cost value.....	13
Fig. 2 - Parental program in genetic programming.....	23
Fig. 3 - Offspring program in genetic programming.....	23
Fig. 4 - Tree structure of final program.....	25
Fig. 5 - Discrete set handling.....	26
Fig. 6 - Principle of the general functional set.....	27
Fig. 7 - Main and simplified principle of AP.....	28
Fig. 8 - AP principle with security principles.....	29
Fig. 9 - Flowchart of the algorithm of evaluation.....	36
Fig. 10 - flowchart of the addstring() function.....	40
Fig. 11 - Flowchart of createText() function.....	41
Fig. 12 - Variability of CV of SOMA and DE for Quintic problem.....	51
Fig. 13 - Variability of CV of SOMA and DE for Sextic problem.....	51
Fig. 14 - Variability of cost value of SOMA and DE for Sinus three problem.....	52
Fig. 15 - Variability of cost values of SOMA and DE for Sinus four problem.....	52
Fig. 16 - Simulation results: SOMA All to One, Quintic problem.....	80
Fig. 17 - Histogram of cost values. SOMA All to one, Quintic problem.....	80
Fig. 18 - Simulation results: SOMA All to all, Quintic problem.....	81
Fig. 19 - Histogram of cost values, SOMA All to all, Quintic problem.....	81
Fig. 20 - Simulation results: Differential evolution, Quintic problem.....	82
Fig. 21 - Histogram of cost values, Differential evolution, Quintic problem.....	82
Fig. 22 - Simulation results: SOMA All to one, Sextic problem.....	83
Fig. 23 - Histogram of cost values - SOMA All to one, Sextic problem.....	83
Fig. 24 - Simulation results: SOMA All to all, Sextic problem.....	84
Fig. 25 - Histogram of cost values: SOMA All to all, Sextic problem.....	84
Fig. 26 - Simulation results: Differential evolution, Sextic problem.....	85
Fig. 27 - Histogram of cost values: Differential evolution, Sextic problem.....	85
Fig. 28 - Simulation results: SOMA All to one, Sinus Three problem.....	86
Fig. 29 - Histogram of cost values: SOMA All to one, Sinus Three problem.....	86
Fig. 30 -Simulation results: SOMA All to all, Sinus Three problem.....	87
Fig. 31 - Histogram of cost values: SOMA All to all, Sinus Three problem.....	87

Fig. 32 -Simulation results: Differential evolution, Sinus Three problem.....	88
Fig. 33 - Histogram of cost values: Differential evolution, Sinus Three problem.....	88
Fig. 34 - Simulation results: SOMA All to one, Sinus Four problem.....	89
Fig. 35 - Histogram of cost values: SOMA All to one, Sinus Four problem.....	89
Fig. 36 - Simulation results: SOMA All to all, Sinus Four problem.....	90
Fig. 37 - Histogram of cost values: SOMA All to all, Sinus Four problem.....	90
Fig. 38 - Simulation results: Differential evolution, Sinus Four problem.....	91
Fig. 39 - Histogram of cost values: Differential evolution, Sinus Four problem.....	91

LIST OF TABLES

Table no. I example of population.....	12
Table no. II Perturbation vector example with 5 parameters and $PRT=0.4$	16
Table no. III: GE grammar in artificial example.....	24
Table no. IV: Chromosome transformation.....	25
Table no. V The structure of input data file data.txt.....	33

THE LIST OF ATTACHMENTS

P1 : Source code dumps

P2 : Simulations results in graphs

Attachments

Attachment no 1. – Dumps of the source codes

Dump no. 12 - Function adding strings

```
void addstring(char* what, char* where, int which)
{
    int datalg=1024;
    char temp[datalg];
    int whr;
    char fce[]="+-*/";
    switch (which)
    {
        case 0: strcpy(temp,what); // before string /Mathematica functions
                strcat(temp,"(");
                strcat(temp, where);
                strcat(temp, ")");
                strcpy(where,temp);
                break;
        case 1: strcpy(temp,"("); // after string /mathematical functions
                strcat(temp,where);
                strcat(temp, ")");
                strcat(temp, what);
                strcpy(where,temp);
                break;
        case 2: strcpy(temp,what); // add constant
                strcpy(where,temp);
                break;
        case 3: strcpy(temp,what); // simply add string before the string
                strcat(temp,"");
                strcat(temp, where);
                strcat(temp, "");
                strcpy(where,temp);
                break;
        default: whr=atoi(what); // to a selected place
                where[which]=fce[whr];
                break;
    }
}
```

Dump no. 13 - spocti() function

```
double spocti(int pozice, int indiv)
{
    double v1,v2;
    int value;
    static int pos;
    static int rezerva;
    value=(int)getPopulation(pozice,indiv);
    switch(value) {
        case 0: if ((PD-1)-(pozice+rezerva)<2)
                { value=value+c2p+1; }
                else
                {
                    pos=pozice+1;
                    rezerva=rezerva+1;
                }
    }
```

```

        v1=spocti(pos, indiv);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        str_penal=str_penal+0.5;
        return (v1+v2);
    }
    break;
case 1: if ((PD-1)-(pozice+rezerva)<2)
    { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=spocti(pos, indiv);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        str_penal=str_penal+0.5;
        return (v1-v2);
    }
    break;
case 2: if ((PD-1)-(pozice+rezerva)<2)
    { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=spocti(pos, indiv);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        str_penal=str_penal+0.5;
        return (v1*v2);
    }
    break;
case 3: if ((PD-1)-(pozice+rezerva)<2)
    {
        value=value+c2p+1;
    }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=spocti(pos, indiv);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        if (v2==0) { return 10000000; break; }
        else { str_penal=str_penal+0.5; return (v1/v2); }
    }
    break;
case 4: if ((PD-1)-(pozice+rezerva)<2)
    { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;

```

```

        v1=spocti(pos, indiv);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        str_penal=str_penal+0.5;
        return (pow(v1,v2));
    }
    break;
}

switch(value) {
    case 5: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=spocti(pos, indiv);
            str_penal=str_penal+0.5;
            return (fabs(v1));
        }
        break;
    case 6: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=spocti(pos, indiv);
            str_penal=str_penal+0.5;
            return (exp(v1));
        }
        break;
    case 7: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=spocti(pos, indiv);
            if (v1<=0) return 10000000; //break;
            str_penal=str_penal+0.5;
            return (log(v1));
        }
        break;
    case 8: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=spocti(pos, indiv);
            if (v1<=0) return 10000000; //break;
            str_penal=str_penal+0.5;
            return (log10(v1));
        }
        break;
    case 9: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }

```

```

        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            if (vl<0) return 10000000; //break;
            str_penal=str_penal+0.5;
            return (sqrt(vl));
        }
        break;
case 10: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            str_penal=str_penal+0.5;
            return (sin(vl));
        }
        break;
case 11: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            str_penal=str_penal+0.5;
            return (cos(vl));
        }
        break;
case 12: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            str_penal=str_penal+0.5;
            return (tan(vl));
        }
        break;
case 13: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            str_penal=str_penal+0.5;
            return (1/(tan(vl)));
        }
        break;
    }

switch(value) {
    case 14: pos=pozice;
            str_penal=str_penal+0.5;

```

```
        return 0;
        break;
case 15: pos=pozice;
        str_penal=str_penal+0.5;
        return 1;
        break;
case 16: pos=pozice;
        str_penal=str_penal+0.5;
        return 2;
        break;
case 17: pos=pozice;
        str_penal=str_penal+0.5;
        return 3;
        break;
case 18: pos=pozice;
        str_penal=str_penal+0.5;
        return 4;
        break;
case 19: pos=pozice;
        str_penal=str_penal+0.5;
        return 5;
        break;
case 20: pos=pozice;
        str_penal=str_penal+0.5;
        return 6;
        break;
case 21: pos=pozice;
        str_penal=str_penal+0.5;
        return 7;
        break;
case 22: pos=pozice;
        str_penal=str_penal+0.5;
        return 8;
        break;
case 23: pos=pozice;
        str_penal=str_penal+0.5;
        return 9;
        break;
case 24: pos=pozice;
        str_penal=str_penal+0.5;
        return (InitVal+(krok*SampleFreq));
        break;
case 25: pos=pozice;
        str_penal=str_penal+0.5;
        return (InitVal+(krok*SampleFreq));
        break;
case 26: pos=pozice;
        str_penal=str_penal+0.5;
        return (InitVal+(krok*SampleFreq));
        break;
case 27: pos=pozice;
        str_penal=str_penal+0.5;
        return M_PI;
        break;
```

```

    }
}

```

Dump no. 14 - createtext() function

```

char* createtext(int pozice, int indiv)
{
    double v1,v2;           // variables for computed values
    char s1[1024];          // string variables
    char s2[1024];
    char s3[1024];
    s1[0]='\0';             // initialisation => empty strings
    s2[0]='\0';
    s3[0]='\0';
    int value;
    static int pos;
    static int rezerva;
    value=(int)getIntPopulation(pozice,indiv);
    switch(value) {
        case 0: if ((PD-1)-(pozice+rezerva)<2)
            { value=value+c2p+1; }
            else
            {
                pos=pozice+1;
                rezerva=rezerva+1;
                v1=spocti(pos, indiv);
                addstring(createtext(pos, indiv),s1,3);
                rezerva=rezerva-1;
                v2=spocti(pos+1, indiv);
                addstring(createtext(pos+1, indiv),s2,3);
                strcpy(TextVec,s1);
                strcat(TextVec,"+");
                strcat(TextVec,s2);
                return (TextVec);
            }
            break;
        case 1: if ((PD-1)-(pozice+rezerva)<2)
            { value=value+c2p+1; }
            else
            {
                pos=pozice+1;
                rezerva=rezerva+1;
                v1=spocti(pos, indiv);
                addstring(createtext(pos, indiv),s1,3);
                rezerva=rezerva-1;
                v2=spocti(pos+1, indiv);
                addstring(createtext(pos+1, indiv),s2,3);
                strcpy(TextVec,s1);
                strcat(TextVec,"-");
                strcat(TextVec,s2);
                return (TextVec);
            }
            break;
        case 2: if ((PD-1)-(pozice+rezerva)<2)
            { value=value+c2p+1; }
            else

```

```

    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=spocti(pos, indiv);
        addstring(createtext(pos, indiv),s1,3);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        addstring(createtext(pos+1, indiv),s2,3);
        strcpy(TextVec,"(");
        strcat(TextVec,s1);
        strcat(TextVec,")");
        strcat(TextVec,"*");
        strcat(TextVec,"(");
        strcat(TextVec,s2);
        strcat(TextVec,")");
        return (TextVec);
    }
    break;
case 3: if ((PD-1)-(pozice+rezerva)<2)
    { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=spocti(pos, indiv);
        addstring(createtext(pos, indiv),s1,3);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        if (v2==0) { strcpy(TextVec, s1); return TextVec; break; }
        else {
            addstring(createtext(pos+1, indiv),s2,3);
            strcpy(TextVec,"(");
            strcat(TextVec,s1);
            strcat(TextVec,")");
            strcat(TextVec,"/");
            strcat(TextVec,"(");
            strcat(TextVec,s2);
            strcat(TextVec,")");
            //printf("v1/v2= %s\n",TextVec);
            return (TextVec);
        }
    }
    break;
case 4: if ((PD-1)-(pozice+rezerva)<2)
    { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=spocti(pos, indiv);
        addstring(createtext(pos, indiv),s1,3);
        rezerva=rezerva-1;
        v2=spocti(pos+1, indiv);
        addstring(createtext(pos+1, indiv),s2,3);

```

```

        strcpy(TextVec, "(");
        strcat(TextVec, s1);
        strcat(TextVec, ")");
        strcat(TextVec, "^");
        strcat(TextVec, "(");
        strcat(TextVec, s2);
        strcat(TextVec, ")");
        return (TextVec);
    }
    break;
}

switch(value) {
    case 5: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            strcpy(s1, createtext(pos, indiv));
            addstring("Abs", s1, 0);
            strcpy(TextVec, s1);
            return (TextVec);
        }
        break;
    case 6: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            strcpy(s1, createtext(pos, indiv));
            addstring("Exp", s1, 0);
            strcpy(TextVec, s1);
            return (TextVec);
        }
        break;
    case 7: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            vl=spocti(pos, indiv);
            strcpy(s1, createtext(pos, indiv));
            if (vl>0) {
                addstring("Log", s1, 0);
                strcpy(TextVec, s1);
                return (TextVec);
            }
            strcpy(TextVec, s1);
            return (TextVec);
        }
        break;
    case 8: if ((PD-1)-(pozice+rezerva)<1)

```

```
{ value=value+clp; }
else
{
pos=pozice+1;
v1=spocti(pos, indiv);
strcpy(s1,createtext(pos, indiv));
if (v1>0) {
    strcpy(s3,"10,");
    strcat(s3,s1);
    addstring("Log10",s1,0);
    strcpy(TextVec, s1);
    return (TextVec);
}
strcpy(TextVec, s1);
return (TextVec);
}
break;
case 9: if ((PD-1)-(pozice+rezerva)<1)
{ value=value+clp; }
else
{
pos=pozice+1;
v1=spocti(pos, indiv);
strcpy(s1,createtext(pos, indiv));
if (v1>=0) {
    addstring("Sqrt",s1,0);
    strcpy(TextVec, s1);
    return (TextVec);
}
strcpy(TextVec, s1);
return (TextVec);
}
break;
case 10: if ((PD-1)-(pozice+rezerva)<1)
{ value=value+clp; }
else
{
pos=pozice+1;
v1=spocti(pos, indiv);
strcpy(s1,createtext(pos, indiv));
addstring("Sin",s1,0);
strcpy(TextVec, s1);
return (TextVec);
}
break;
case 11: if ((PD-1)-(pozice+rezerva)<1)
{ value=value+clp; }
else
{
pos=pozice+1;
v1=spocti(pos, indiv);
strcpy(s1,createtext(pos, indiv));
addstring("Cos",s1,0);
strcpy(TextVec, s1);
```

```
        return (TextVec);
    }
    break;
case 12: if ((PD-1)-(pozice+rezerva)<1)
    { value=value+clp; }
    else
    {
        pos=pozice+1;
        vl=spocti(pos, indiv);
        strcpy(s1,createtext(pos, indiv));
        addstring("Tan",s1,0);
        strcpy(TextVec, s1);
        return (TextVec);
    }
    break;
case 13: if ((PD-1)-(pozice+rezerva)<1)
    { value=value+clp; }
    else
    {
        pos=pozice+1;
        vl=spocti(pos, indiv);
        strcpy(s1,createtext(pos, indiv));
        addstring("Cot",s1,0);
        strcpy(TextVec, s1);
        return (TextVec);
    }
    break;
}

switch(value) {
    case 14: pos=pozice;
        return "0";
        break;
    case 15: pos=pozice;
        return "1";
        break;
    case 16: pos=pozice;
        return "2";
        break;
    case 17: pos=pozice;
        return "3";
        break;
    case 18: pos=pozice;
        return "4";
        break;
    case 19: pos=pozice;
        return "5";
        break;
    case 20: pos=pozice;
        return "6";
        break;
    case 21: pos=pozice;
        return "7";
        break;
}
```

```

        case 22: pos=pozice;
                return "8";
                break;
        case 23: pos=pozice;
                return "9";
                break;
        case 24: pos=pozice;
                return "x";
                break;
        case 25: pos=pozice;
                return "x";
                break;
        case 26: pos=pozice;
                return "x";
                break;
        case 27: pos=pozice;
                return "PI";
                break;
    }
}

```

Dump no. 15 - analyzuj() function

```

double analyzuj(int pozice, int indiv)
{
    double v1,v2;
    int value;
    static int pos;
    static int rezerva;
    static int pocet_konstant;
    value=(int)getIntPopulation(pozice,indiv);
    switch(value) {
        case 0: if ((PD-1)-(pozice+rezerva)<2)
                { value=value+c2p+1; }
                else
                {
                    pos=pozice+1;
                    rezerva=rezerva+1;
                    v1=analyzuj(pos, indiv);
                    rezerva=rezerva-1;
                    v2=analyzuj(pos+1, indiv);
                    return (v1+v2);
                }
                break;
        case 1: if ((PD-1)-(pozice+rezerva)<2)
                { value=value+c2p+1; }
                else
                {
                    pos=pozice+1;
                    rezerva=rezerva+1;
                    v1=analyzuj(pos, indiv);
                    rezerva=rezerva-1;
                    v2=analyzuj(pos+1, indiv);
                    return (v1-v2);
                }
                break;
        case 2: if ((PD-1)-(pozice+rezerva)<2)

```

```

        { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=analyzuj(pos, indiv);
        rezerva=rezerva-1;
        v2=analyzuj(pos+1, indiv);
        return (v1*v2);
    }
    break;
case 3: if ((PD-1)-(pozice+rezerva)<2)
    { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=analyzuj(pos, indiv);
        rezerva=rezerva-1;
        v2=analyzuj(pos+1, indiv);
        if (v2==0) { return v1; break; }
        else { return (v1/v2); }
    }
    break;
case 4: if ((PD-1)-(pozice+rezerva)<2)
    { value=value+c2p+1; }
    else
    {
        pos=pozice+1;
        rezerva=rezerva+1;
        v1=analyzuj(pos, indiv);
        rezerva=rezerva-1;
        v2=analyzuj(pos+1, indiv);
        return (pow(v1,v2));
    }
    break;
}

switch(value) {
    case 5: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=analyzuj(pos, indiv);
            return (fabs(v1));
        }
        break;
    case 6: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=analyzuj(pos, indiv);
            return (exp(v1));
        }
}

```

```
        break;
case 7: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=analyzuj(pos, indiv);
            if (v1<=0) return v1; //break;
            return (log(v1));
        }
        break;
case 8: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=analyzuj(pos, indiv);
            if (v1<=0) return v1; //break;
            return (log10(v1));
        }
        break;
case 9: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=analyzuj(pos, indiv);
            if (v1<0) return v1; //break;
            return (sqrt(v1));
        }
        break;
case 10: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=analyzuj(pos, indiv);
            return (sin(v1));
        }
        break;
case 11: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
            v1=analyzuj(pos, indiv);
            return (cos(v1));
        }
        break;
case 12: if ((PD-1)-(pozice+rezerva)<1)
        { value=value+clp; }
        else
        {
            pos=pozice+1;
```

```

        v1=analyzuj(pos, indiv);
        return (tan(v1));
    }
    break;
case 13: if ((PD-1)-(pozice+rezerva)<1)
    { value=value+clp; }
    else
    {
        pos=pozice+1;
        v1=analyzuj(pos, indiv);
        return (1/(tan(v1)));
    }
    break;
}

switch(value) {
    case 14: pos=pozice;
        pocet_konstant=pocet_konstant+1;
        Constant=1.1;
        konstanty=konstanty+1;
        return (Constant);
        break;
    case 15: pos=pozice;
        return (InitVal+(krok*SampleFreq));
        break;
    case 16: pos=pozice;
        return M_PI;
        break;
}
}

```

Dump no. 16 - Data loading and memory structures creation

```

// Loading data for regression
strcpy(InDataFile,"data.txt");
if ((data_f = fopen(InDataFile, "rt")) == NULL)
{
    printf("Cannot open data file !!!\n");
    goto END;
}
// read the data file
fscanf(data_f, "%d",&DataLength);
fscanf(data_f, "%f",&SampleFreq);
fscanf(data_f, "%f",&InitVal);
// print the values read from the file
printf("\n Data length %d",DataLength);
printf("\n Sample frequency %f",SampleFreq);
printf("\n Initial Value %f",InitVal);
// load data from the file to the DataVec
createDataVec(DataLength);
createFinalVec(DataLength);
displayDataVec(DataLength);
fclose(data_f);

```

Dump no. 17. - condition at the end of evaluation code

```
if (Mpocet == 0)
    // saving to popualtion
    { setPopulation (PD - 1, Indiv, CValue); }
else
    // saving to second evolution
    { setPop (MPD - 1, Indiv, CValue); }
```

Attachment no 2. – Simulations results in graphs

Simulation graphs of results gained during simulation experiments follows.

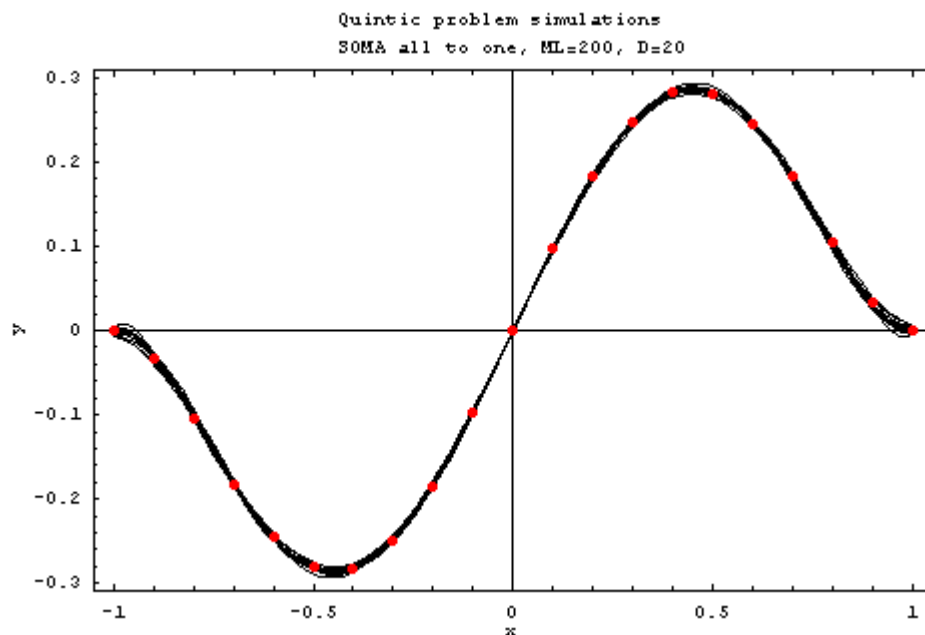


Fig. 16 - Simulation results: SOMA All to One, Quintic problem

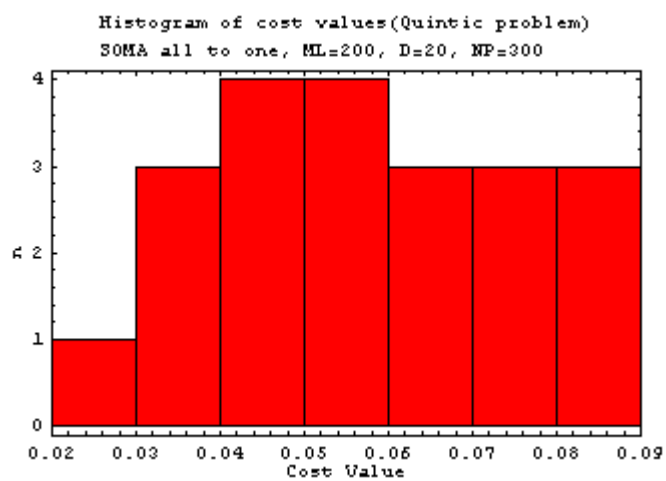


Fig. 17 - Histogram of cost values. SOMA All to one, Quintic problem

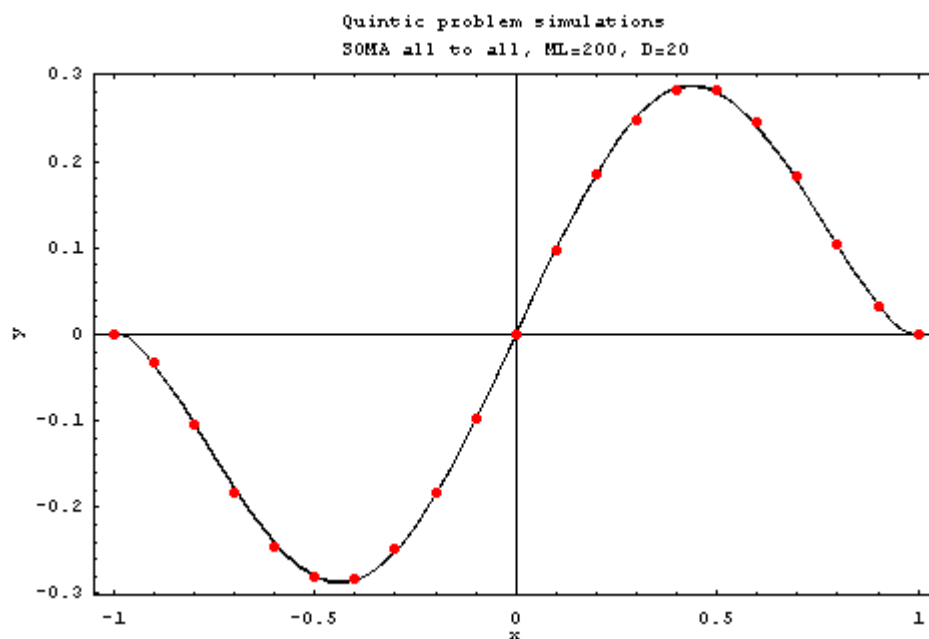


Fig. 18 - Simulation results: SOMA All to all, Quintic problem

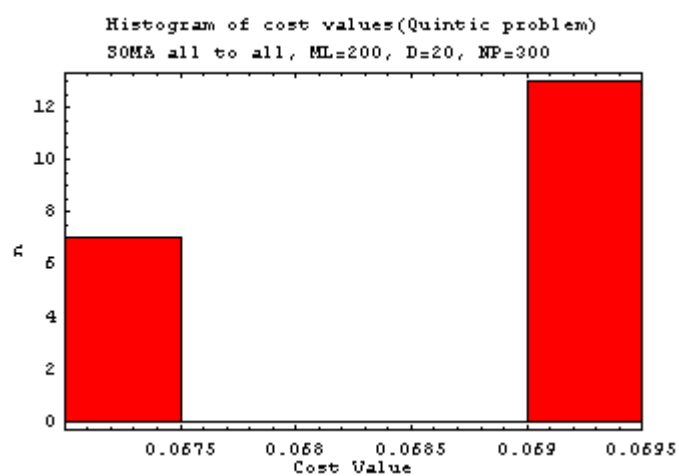


Fig. 19 - Histogram of cost values, SOMA All to all, Quintic problem

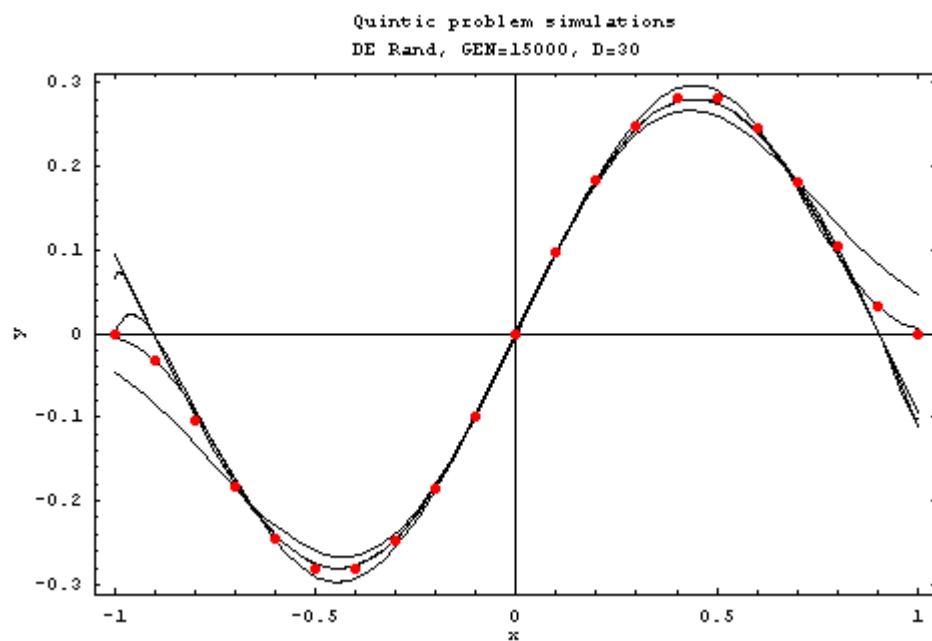


Fig. 20 - Simulation results: Differential evolution, Quintic problem

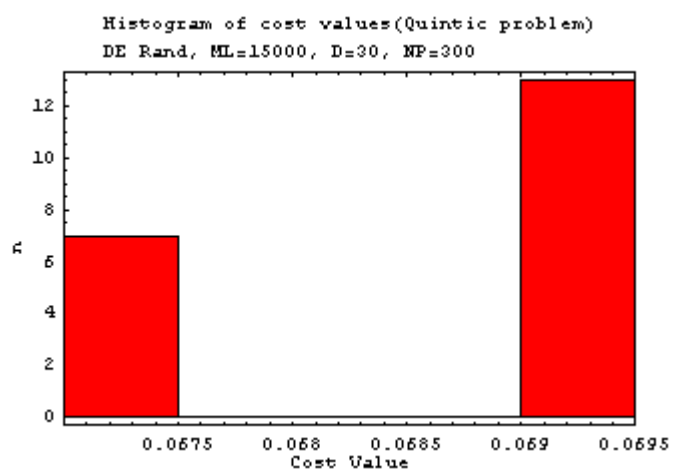


Fig. 21 - Histogram of cost values, Differential evolution, Quintic problem

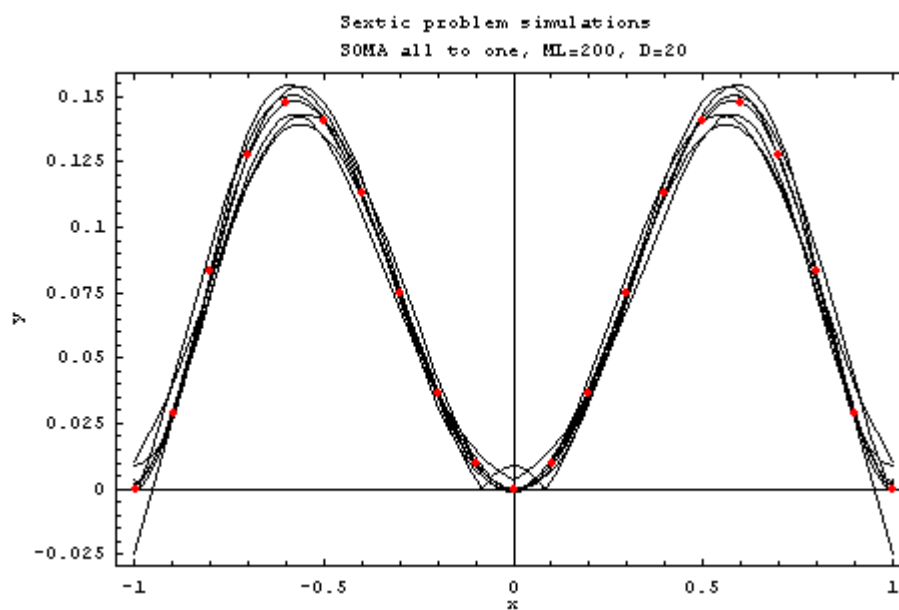


Fig. 22 - Simulation results: SOMA All to one, Sextic problem

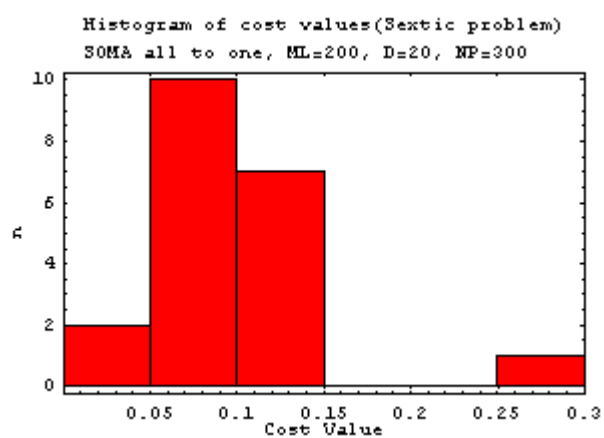


Fig. 23 - Histogram of cost values - SOMA All to one, Sextic problem

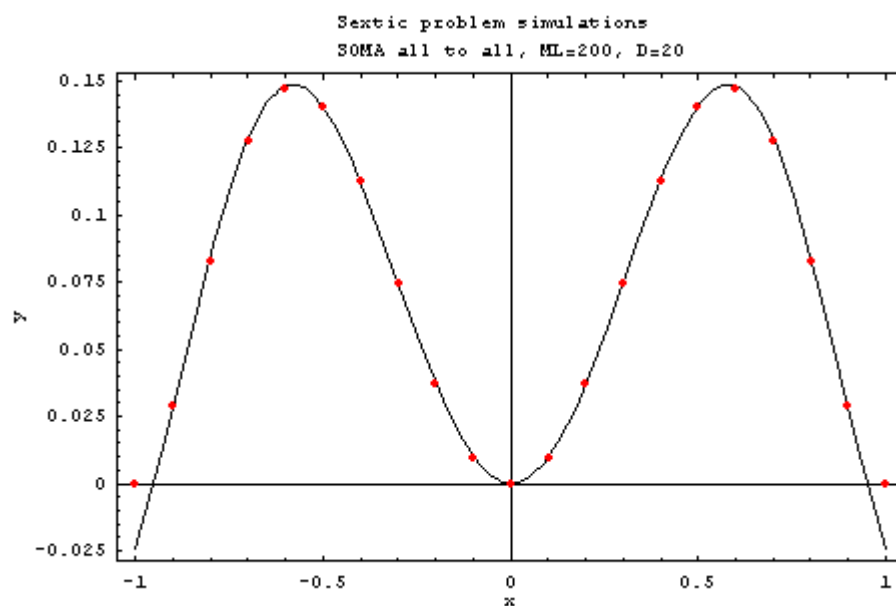
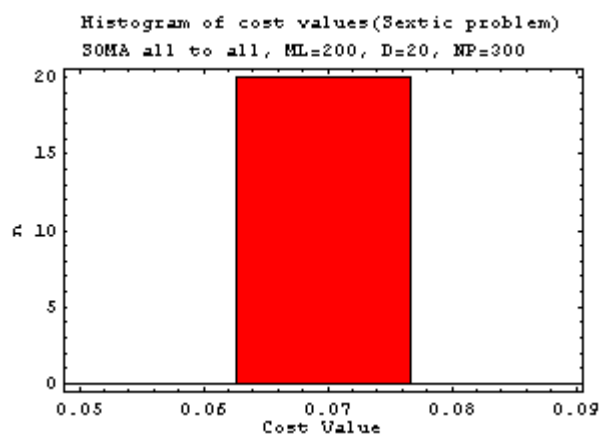


Fig. 24 - Simulation results: SOMA All to all, Sextic problem

Fig. 25 - Histogram of cost values: SOMA All to all,
Sextic problem

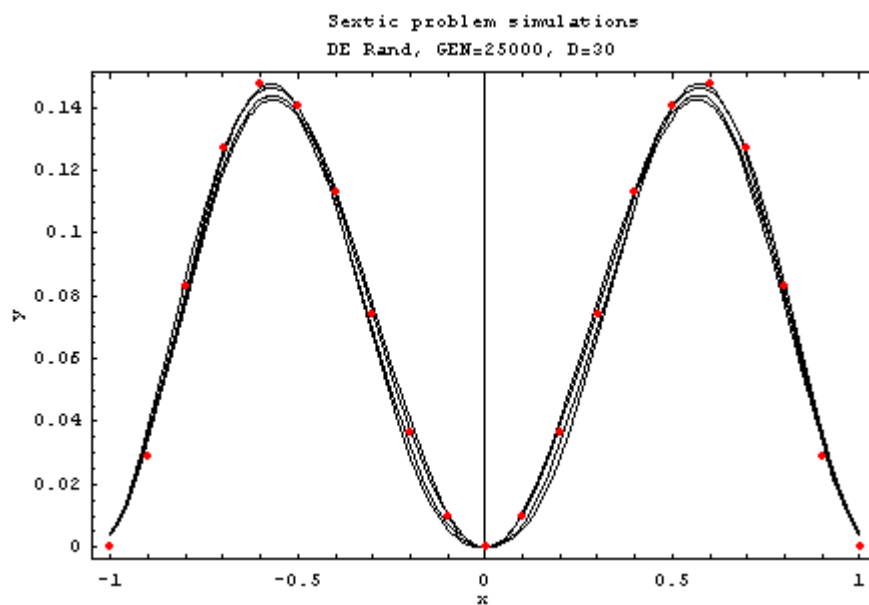


Fig. 26 - Simulation results: Differential evolution, Sextic problem

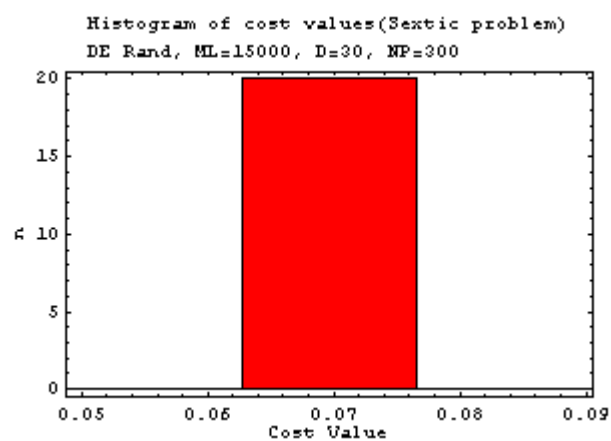


Fig. 27 - Histogram of cost values: Differential evolution, Sextic problem

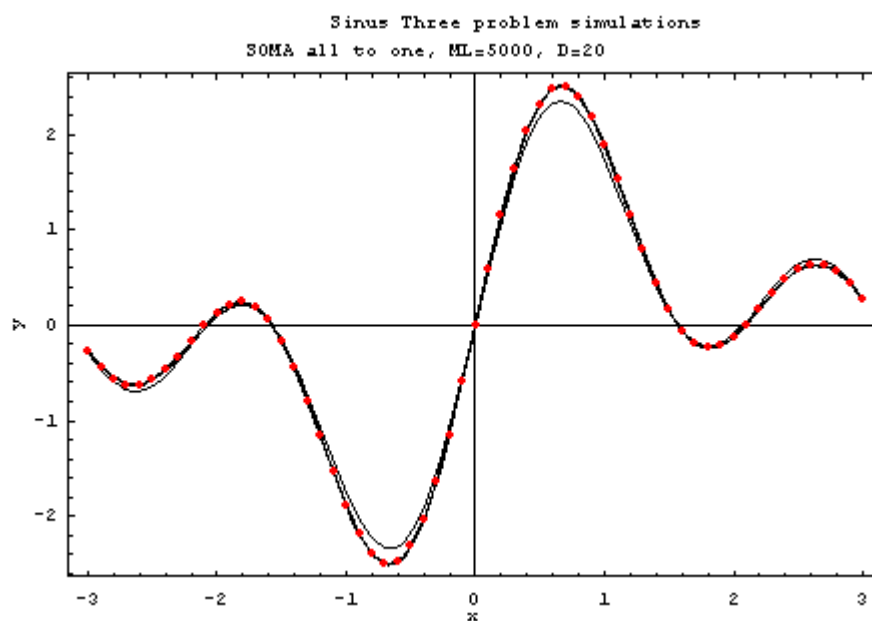


Fig. 28 - Simulation results: SOMA All to one, Sinus Three problem

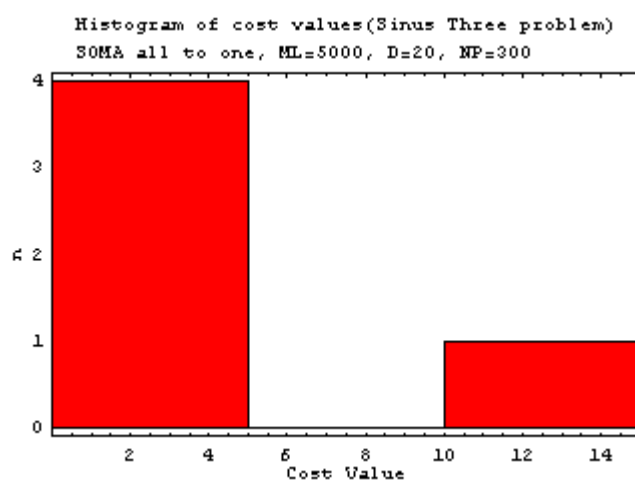


Fig. 29 - Histogram of cost values: SOMA All to one, Sinus Three problem

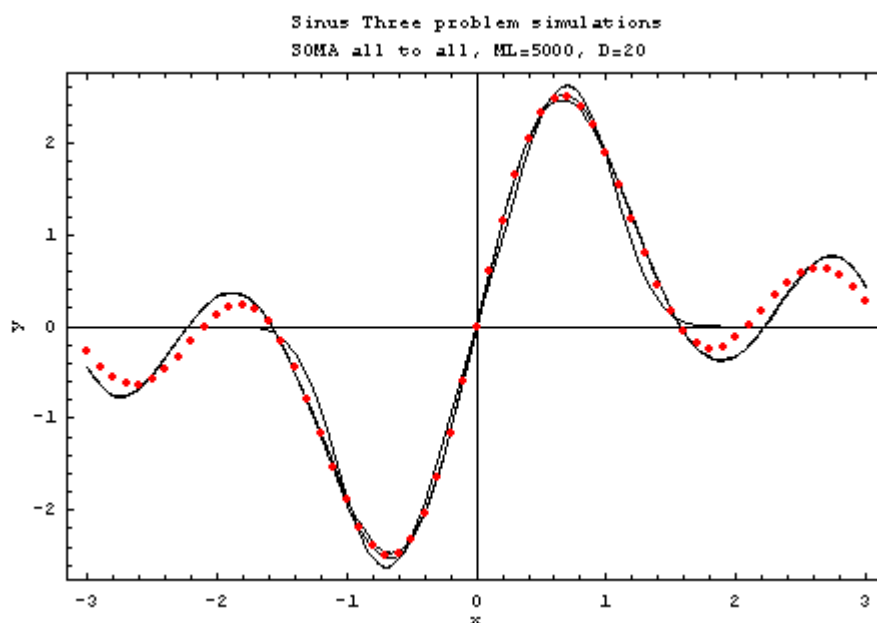


Fig. 30 -Simulation results: SOMA All to all, Sinus Three problem

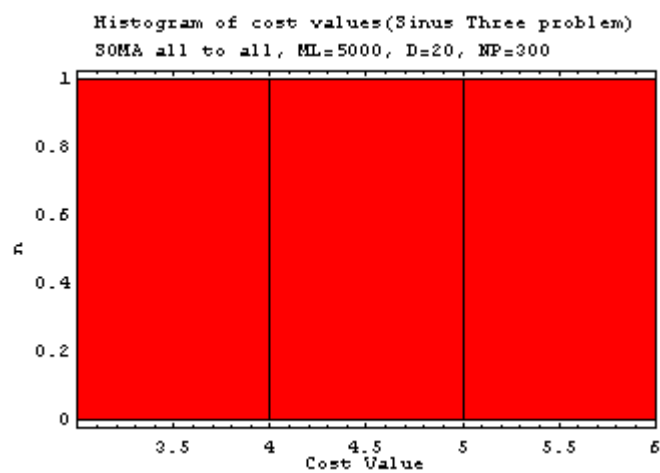


Fig. 31 - Histogram of cost values: SOMA All to all, Sinus Three problem

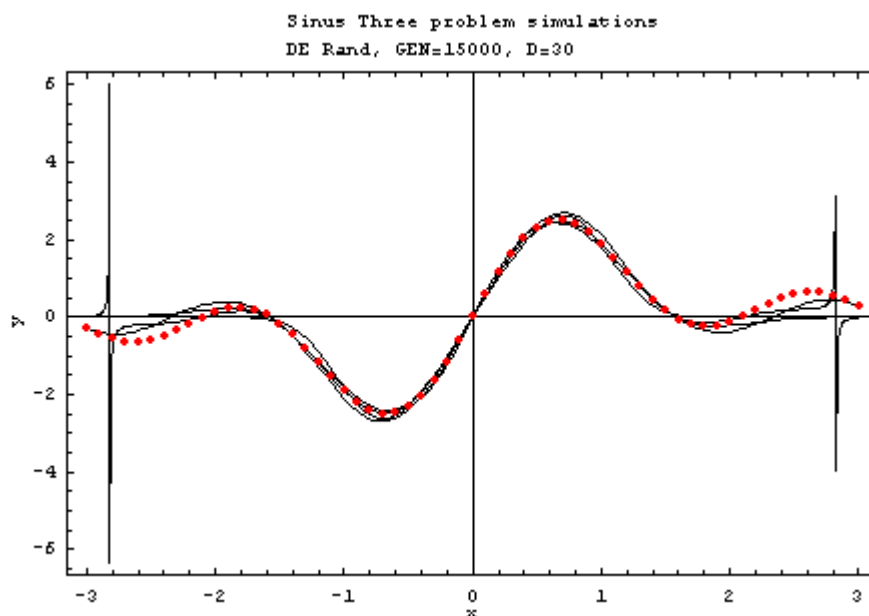


Fig. 32 -Simulation results: Differential evolution, Sinus Three problem

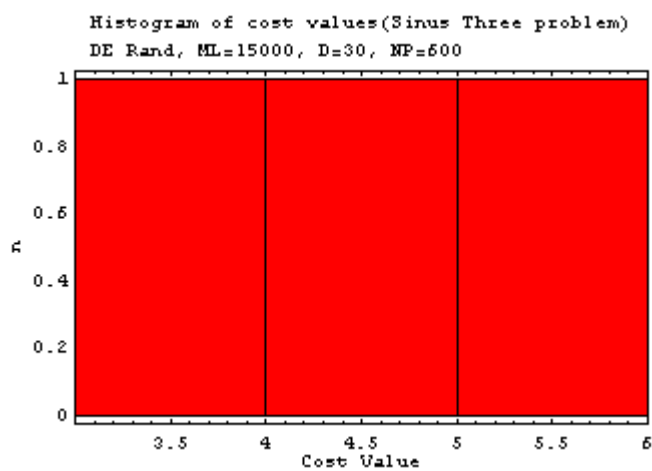


Fig. 33 - Histogram of cost values: Differential evolution, Sinus Three problem

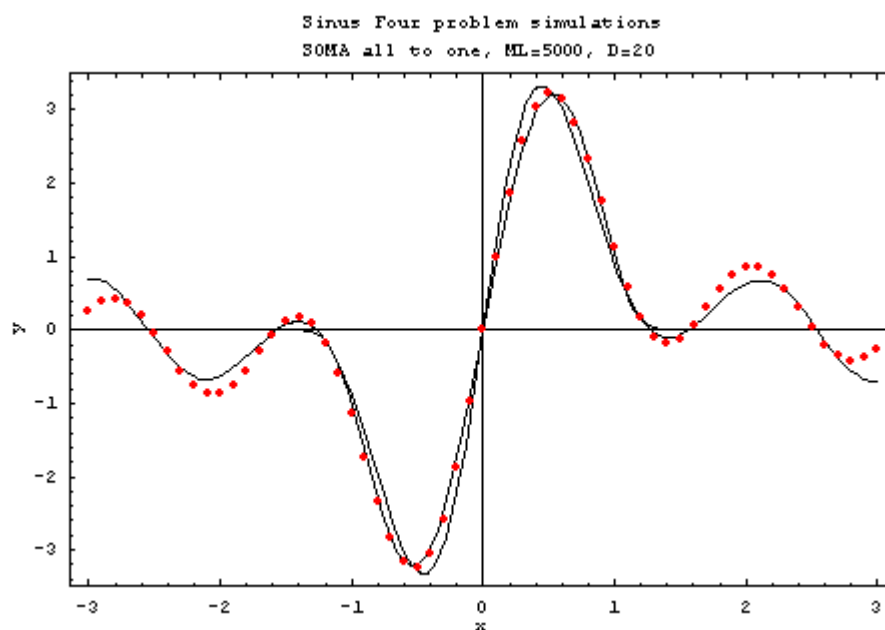
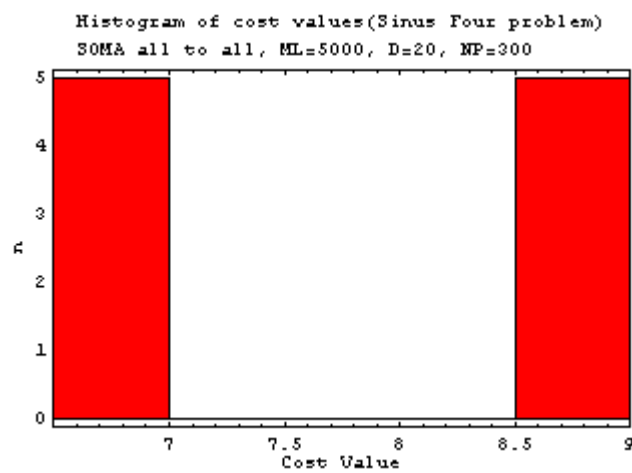


Fig. 34 - Simulation results: SOMA All to one, Sinus Four problem

Fig. 35 - Histogram of cost values: SOMA All to one,
Sinus Four problem

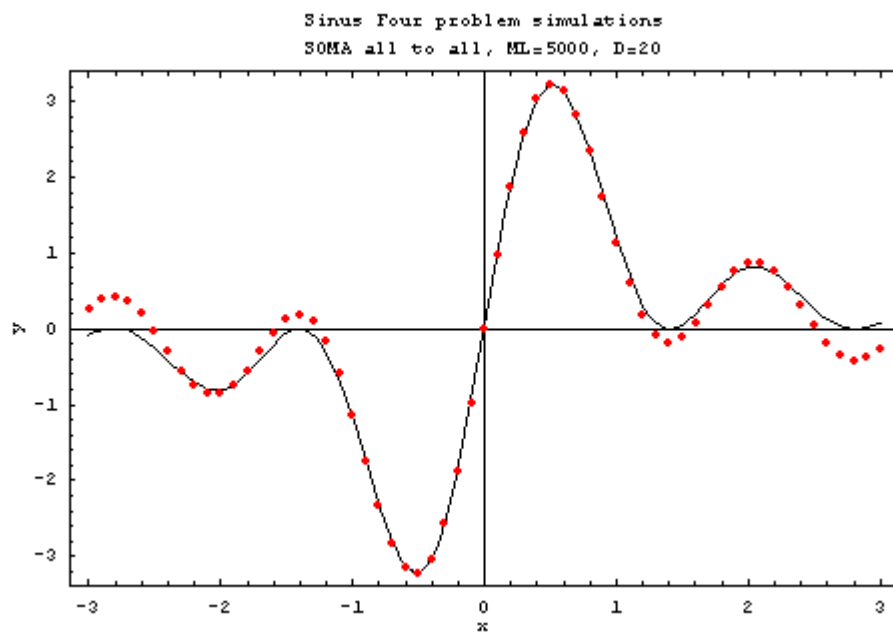
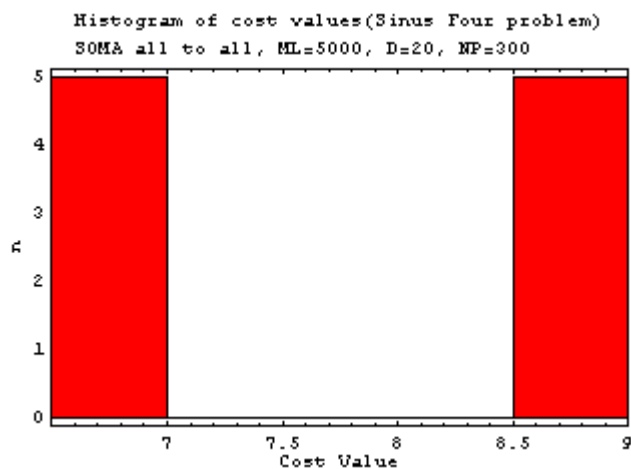


Fig. 36 - Simulation results: SOMA All to all, Sinus Four problem

Fig. 37 - Histogram of cost values: SOMA All to all, Sinus



Four problem

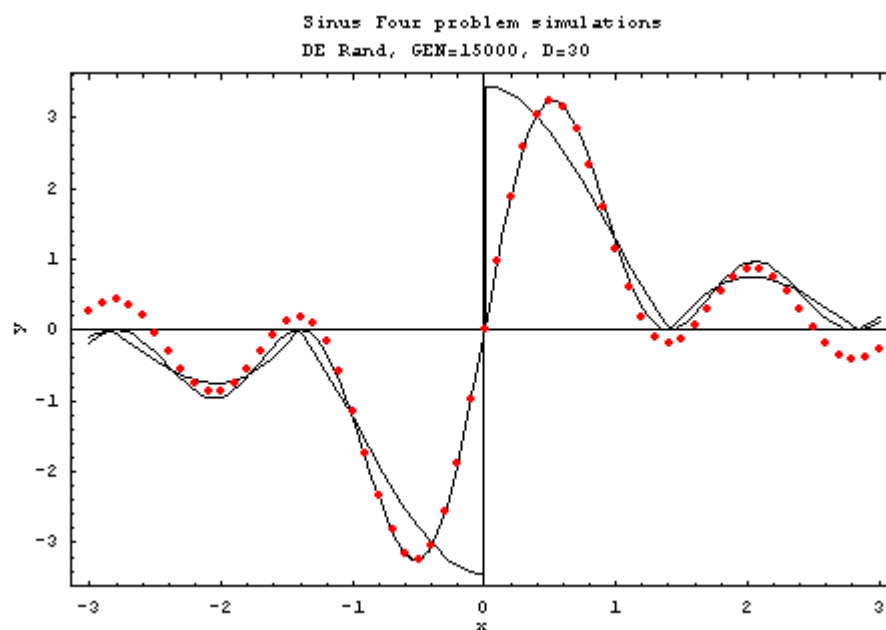
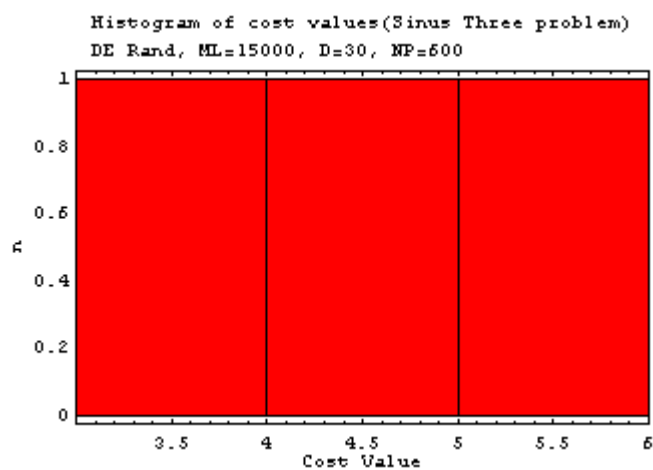


Fig. 38 - Simulation results: Differential evolution, Sinus Four problem

Fig. 39 - Histogram of cost values: Differential evolution,
Sinus Four problem