ANALYTICAL PROGRAMMING WITH EXTENDED INDIVIDUALS

Adam Viktorin
Michal Pluhacek
Zuzana Kominkova Oplatkova
Roman Senkerik
Tomas Bata University in Zlin, Faculty of Applied Informatics
Nam T.G. Masaryka 5555, 760 01 Zlin, Czech Republic
{aviktorin, pluhacek, oplatkova, senkerik}@fai.utb.cz

KEYWORDS
Analytical Programming, Differential Evolution, SHADE, Constant Estimation

ABSTRACT

This paper proposes a new technique for the estimation of values of constants in programs synthesized by Analytical Programming (AP). Proposed technique divides the features of an individual in Evolutionary Algorithm (EA) into two parts – program part and constant part. Features in program part are mapped to synthesized program and features in constant part are used as constant values. AP with implementation of this technique is tested on four benchmark functions – Quintic, Sextic, 3Sine and 4Sine.

INTRODUCTION

Analytical Programming (AP) is a novel approach to symbolic structure synthesis which uses Evolutionary Algorithm (EA) for its computation. It was introduced by Zelinka in 2001 (Zelinka 2001) and since its introduction, it has been proven on numerous problems to be as suitable for symbolic structure synthesis as Genetic Programming (GP) (Koza 1990; Zelinka, Oplatkova 2003; Zelinka et al. 2005; Oplatkova, Zelinka 2006; Zelinka et al. 2008; Senkerik et al. 2013).
AP is based on the set of functions and terminals called General Functional Set (GFS). The individual of an EA is translated from individual domain to program domain using this set. The subset of terminals often contains constants but the proper amount and which ones should be used is dependent on the optimized task. Zelinka et al. (Zelinka et al. 2005) solved this problem by using only one constant *K,* which is indexed in the synthesized formula as K_1, K_2, …, K_n and the values are estimated using secondary EA (meta-evolution) or by non-linear fitting. The first mentioned method is quite time consuming and the time estimation is hard because of the varying number of constants in synthesized programs. The later is also time consuming and depends on the implementation of the non-linear fitting method. Because of these disadvantages a new approach is presented in this paper which extends the individual, thus increasing the dimensionality. The extended features are used for the evolution of the constant values. Therefore, time complexity is increased, but it is easy to estimate it and no secondary evolution process takes place. Furthermore, a novel state-of-art version of Differential Evolution (DE) called Success-History based Adaptive Differential Evolution (SHADE) (Tanabe, Fukunaga 2013) was used to carry out the evolution.
This approach was tested on some of the basic benchmark functions (Koza 1994) and the results are presented below.
The rest of the paper is structured as follows: Next section provides the description of AP, section that follows describes the constant handling process and following section is about DE and SHADE. Experiment setup and results are provided in two following sections and the paper is concluded in the last section.

ANALYTICAL PROGRAMMING

The basic functionality of AP is formed by three parts – General Functional Set (GFS), Discrete Set Handling (DSH) and Security Procedures (SPs). GFS contains all elementary objects which can be used to form a program, DSH carries out the mapping of individuals to programs and SPs are implemented into mapping process to avoid mapping to pathological programs and into cost function to avoid critical situations.

General Functional Set

AP uses sets of functions and terminals. Functions require at least one additional argument for computation, whereas terminals require no arguments and are final (e.g. constants, independent variables). The synthesized program is branched by functions requiring two and more arguments and the length of it is extended by functions which require one argument. Terminals do not contribute to the complexity of the synthesized program (length) but are needed in order to synthesize a non-pathological program (program that can be evaluated by cost function). Therefore, each non-pathological program must contain at least one terminal.
Combined set of functions and terminals forms GFS which is used for mapping from individual domain to program domain. The content of GFS is dependent on user choice. GFS is nested and can be divided into subsets according to the number of arguments that the subset requires. GFS_{0arg} is a subset which requires zero arguments, thus contains only terminals. GFS_{1arg} contains all terminals and functions requiring one

Proceedings 30th European Conference on Modelling and Simulation ©ECMS Thorsten Claus, Frank Herrmann, Michael Manitz, Oliver Rose (Editors)
ISBN: 978-0-9932440-2-5 / ISBN: 978-0-9932440-3-2 (CD)

argument, GFS_{2arg} contains all objects from GFS_{1arg} and functions requiring two arguments and so on, GFS_{all} is a complete set of all elementary objects. The GFS used in the theoretical part of this paper is depicted below and the division into subsets is also shown.

- $\mathbf{GFS_{0arg}}$: x, k
- $\mathbf{GFS_{1arg}}$: sin, cos, x, k
- $\mathbf{GFS_{2arg}} = \mathbf{GFS_{all}}$: +, -, *, /, sin, cos, x, k

For the purpose of mapping from individual to the program, it is important to note that objects in GFS are ordered by a number of arguments they require in descending order.

Discrete Set Handling

DSH is used for mapping the individual to the synthesized program. Most of the EAs use individuals with real numbered features. The first important step in order for DSH to work is to get individual with integer features which is done by rounding real feature values. The integer values of an individual are indexes into the discrete set, in this case, GFS and its subsets. If the index value is greater than the size of used GFS, modulo operation with the size of the discrete set is performed. Two examples of mapping are depicted in (1) and (2).

$$\begin{gathered} Individual = \begin{Bmatrix} 0.12, 4.29, 6.92, 6.12, 2.45, \\ 6.33, 5.78, 0.22, 1.94, 7.32 \end{Bmatrix} \\ Rounded\ individual = \{0, 4, 7, 6, 2, 6, 6, 0, 2, 7\} \\ GFS_{all} = \{+, -, *, /, sin, cos, x, k\} \\ \boldsymbol{Program}: \mathbf{sin}\, \boldsymbol{x} + \boldsymbol{k} \end{gathered} \quad (1)$$

The objects in GFS_{all} are indexed from 0 and mapping is as follows: The first rounded individual feature is 0 which represents + function in GFS_{all}. This function requires two arguments and those are represented by next two indexes – 4 and 7, which are mapped to function *sin* and constant *k*. The *sin* function requires one argument which is given by next index (rounded feature) – 6 and it is mapped to variable *x*. Since there is no possible way of branching the program further, other features are ignored and synthesized program is *sin x + k*.

Mapping steps:

1. Index 0 is mapped to +. Program: _ + _
2. Index 4 is mapped to *sin*. Program: *sin* _ + _
3. Index 7 is mapped to *k*. Program: *sin* _ + *k*
4. Index 6 is mapped to *x*. Program: *sin x + k*
5. Remaining indexes {2, 6, 6, 0, 2, 7} are ignored because the program is complete.

Where _ denotes the space in the program which needs to be filled with objects from GFS.

$$\begin{gathered} Individual = \begin{Bmatrix} 5.08, 1.64, 5.58, 4.41, 6.20, \\ 1.28, 0.07, 3.99, 5.27, 2.64 \end{Bmatrix} \\ Rounded\ individual = \{5, 2, 6, 4, 6, 1, 0, 4, 5, 3\} \\ GFS_{all} = \{+, -, *, /, sin, cos, x, k\} \\ \boldsymbol{Program}: \mathbf{cos}(\boldsymbol{x} * \mathbf{sin}\, \boldsymbol{x}) \end{gathered} \quad (2)$$

The first index to GFS_{all} is 5, which represents *cos* function, its argument is chosen by next index – 2 representing function * which needs two arguments. Arguments are indexed 6 and 4 – variable *x* and function *sin*. In this step only one more argument for function *sin* is needed and it is variable *x* denoted by index 6. The synthesized program is therefore *cos(x * sin x)*.

Mapping steps:

1. Index 5 is mapped to *cos*. Program: *cos* _
2. Index 2 is mapped to *. Program: *cos(_ * _)*
3. Index 6 is mapped to *x*. Program: *cos(x * _)*
4. Index 4 is mapped to *sin*. Program: *cos(x * sin _)*
5. Index 6 is mapped to *x*. Program: *cos(x * sin x)*
6. Remaining indexes {1, 0, 4, 5, 3} are ignored because the program is complete.

It is worthwhile to note that in both examples individual features were not fully used and synthesized programs are not as complex as the dimensionality enables them to be. Moreover, both examples use indexes which are lower than the size of GFS_{all} therefore, no modulo operation is needed.

Security Procedures

SPs are used in AP to avoid critical situations. Some of the SPs are implemented into the AP itself and some have to be implemented into the cost function evaluation. The typical representatives of the later are checking synthesized programs for loops, infinity and imaginary numbers if not expected (dividing by 0, square root of negative numbers, etc.).

The most significant SP implemented in AP is checking for pathological programs. Pathological programs are programs which cannot be evaluated due to the absence of arguments in the synthesized function. For example, individual with rounded features of {4, 4, 4, 4, 4} would be mapped to program *cos(cos(cos(cos(cos_))))* which lacks constant or variable at the empty position denoted by _ and thus represent a pathological program. Such situation can be avoided by a simple procedure which checks how far is the end of the individual during mapping and according to that maps rounded individual features to subsets of GFS_{all} which do not require too many arguments. With the previous example using GFS from GFS section, the mapping process would be as follows:

1. First three features {4, 4, 4} already mapped to GFS_{all}. Program: *cos(cos(cos _))*
2. Current index in rounded individual features is 4 and only 1 feature is left to the end of the individual therefore, the index is mapped to GFS_{1arg} and modulo operation by the size of GFS_{1arg} which is 4 is performed, thus index = 4 *mod* 4 = 0. The index is mapped to *sin*. Program: *cos(cos(cos(sin _)))*
3. Last index in rounded individual is 4 and no features are left to the end of the individual, therefore index is mapped to GFS_{0arg} and modulo operation by the size of GFS_{0arg} which is 2 is performed, thus index = 4 *mod* 2 = 0. Index is mapped to *x*. Program: *cos(cos(cos(sin x)))*

The program is no longer pathological and can be evaluated. This simple SP is able to eliminate the generation of pathological programs and, therefore, improve the performance of AP because all individuals can be evaluated.

CONSTANT HANDLING

The main novelty of this paper is in the constant handling technique used with AP. In the previous work (Zelinka et al. 2005), constant values in synthesized programs were estimated by second EA (meta-heuristic) or by non-linear fitting. This work presents a novel approach, which uses the extended part of the individual in EA for the evolution of constant values.
The important task was to determine, what is the correct size of an extension (3).

$$k = l - floor\big((l-1) \,/\, max\_arg\big) \tag{3}$$

Where k is the maximum number of constants that can appear in the synthesized program (extension) of length l and *max_arg* is the maximum number of arguments needed by functions in GFS. Also the *floor()* is a common floor round function. The final individual dimensionality (length) will be $k+l$ and the example might be:

- Program length $l = 10$
- GFS: {+, -, *, /, *sin*, *cos*, *x*, *k*}
- GFS maximum argument *max_arg = 2*
- Extension size $k = 10 - floor((10\text{-}1) / 2) = 6$
- Dimensionality of the extended individual $k+l = 16$

This means, that the EA will work with individuals of length 16, but only first 10 features will be used for indexing into the GFS and the rest will be used as constant values.
While mapping the individual into a program, the constants are indexed and later replaced by the value from individual. Simple example can be seen in (4). Individual features in bold are the constant values. It is worthwhile to note that only features which are going to be mapped to GFS are rounded and the rest is omitted.

$$\begin{aligned} Individual &= \{\, 5.08, 1.64, 6.72, 1.09, 6.20, \\ &\quad\; 1.28, \mathbf{0.07}, \mathbf{3.99}, \mathbf{5.27}, \mathbf{2.64} \,\} \\ Rounded\ individual &= \{5, 2, 7, 1, 6, 1\} \\ GFS_{all} &= \{+, -, *, /, sin, cos, x, k\} \\ \boldsymbol{Program}&\boldsymbol{:}\ \mathbf{cos}\big(\boldsymbol{k1} * (\boldsymbol{x} - \boldsymbol{k2})\big) \\ \boldsymbol{Replaced}&\boldsymbol{:}\ \mathbf{cos}\big(\mathbf{0.07} * (\boldsymbol{x} - \mathbf{3.99})\big) \end{aligned} \tag{4}$$

The first index to GFS_{all} is 5, which represents *cos* function, its argument is chosen by next index – 2 representing function * which needs two arguments. Arguments are indexed 7 and 1 – constant *k1* and function -. After this step, two arguments are needed and only two features are left in the program part of the individual. Therefore, the security procedure takes place and those last two features are indexed into GFS_{0arg}. Thus indexes 6 and 1 are mapped to variable *x* (6 mod size(GFS_{0arg}) = 0) and constant *k2*. The synthesized program is therefore *cos(k1 * (x – k2))*. The constants are replaced by the remaining features 0.07 and 3.99 respectively.
Individual mapping steps:

1. Index 5 is mapped to *cos*. Program: *cos _*
2. Index 2 is mapped to *. Program: *cos(_ * _)*
3. Index 7 is mapped to *k1*. Program: *cos(k1 * _)*
4. Index 1 is mapped to -. Program: *cos(k1 * (_ - _))*
5. Index 6 mod 2 = 0 is mapped to *x*. Program: *cos(k1 * (x - _))*
6. Index 1 is mapped to *k2*. Program: *cos(k1 * (x – k2))*
7. Constants are replaced by the remaining features according to their indexes – *k1* is replaced by the first available feature 0.07 and *k2* is replaced by the second available feature 3.99. Program: *cos(0.07 * (x – 3.99))*

SUCCESS-HISTORY BASED ADAPTIVE DIFFERENTIAL EVOLUTION

This section describes the basics of DE and SHADE algorithms.
DE algorithm has three control parameters – population size *NP*, crossover rate *CR* and scaling factor *F*. In the canonical form of DE, those three parameters are static and depend on the user setting. Other important features of DE algorithm are mutation strategy and crossover strategy. Canonical DE uses "rand/1/bin" mutation strategy (5) and binomial crossover (8). SHADE algorithm, on the other hand, uses only two control parameters – population size *NP* and size of historical memories *H*. *F* and *CR* parameters are automatically adapted based on the evolutionary process and its values for each individual are generated according to (7) and (9) respectively. Also, the mutation strategy is different than that of canonical DE. Novel mutation strategy used in SHADE is called "current-to-*p*best/1" and it is depicted in (6). The concept of basic operations in DE and SHADE algorithms is shown in following sections, for a detailed description on feature constraint correction, update of historical memories and external archive handling in SHADE see (Tanabe, Fukunaga 2013).

Mutation Strategies and Parent Selection

In canonical forms of both algorithms, parent vectors are selected by classic PRNG with uniform distribution. Mutation strategy "rand/1/bin" uses three random parent vectors with indexes *r1*, *r2* and *r3*, where $r1 = U[1, NP]$, $r2 = U[1, NP]$, $r3 = U[1, NP]$ and $r1 \neq r2 \neq r3$. Mutated vector $\boldsymbol{v}_{i,\,G}$ is obtained from three different vectors $\boldsymbol{x}_{r1}$, $\boldsymbol{x}_{r2}$, $\boldsymbol{x}_{r3}$ from current generation *G* with help of static scaling factor $F_i = F$ as follows:

$$\boldsymbol{v}_{i,G} = \boldsymbol{x}_{r1,G} + F_i\big(\boldsymbol{x}_{r2,G} - \boldsymbol{x}_{r3,G}\big) \tag{5}$$

Contrarily, SHADEs mutation strategy "current-to-*p*best/1" uses four parent vectors – current *i*-th vector

$\boldsymbol{x}_{i,G}$, vector $\boldsymbol{x}_{pbest,G}$ randomly selected from $NP \times p$ ($p = U[p_{min}, 0.2]$, $p_{min} = 2/NP$) best vectors (in terms of objective function value) from G, randomly selected vector $\boldsymbol{x}_{r1,G}$ from G and randomly selected vector $\boldsymbol{x}_{r2,G}$ from the union of G and external archive A. Where $\boldsymbol{x}_{i,G} \neq \boldsymbol{x}_{r1,G} \neq \boldsymbol{x}_{r2,G}$.

$$\boldsymbol{v}_{i,G} = \boldsymbol{x}_{i,G} + F_i\left(\boldsymbol{x}_{pbest,G} - \boldsymbol{x}_{i,G}\right) + F_i\left(\boldsymbol{x}_{r1,G} - \boldsymbol{x}_{r2,G}\right) \quad (6)$$

The scaling factor F_i is generated from Cauchy distribution with location parameter value of $M_{F,r}$ which is randomly selected value from scale factor historical memory, and scale parameter value of 0.1 (7).

$$F_i = C\left[M_{F,r}, 0.1\right] \quad (7)$$

Crossover and Elitism

The trial vector $\boldsymbol{u}_{i,G}$ which is compared with original vector $\boldsymbol{x}_{i,G}$ is completed by crossover operation (8) and this operation is the same for both DE and SHADE algorithms. CR_i value in DE algorithm is again static $CR_i = CR$ whereas with SHADE algorithm its value is generated from a normal distribution with a mean parameter value of $M_{CR,r}$ which is randomly selected value from crossover rate historical memory and with standard deviation value of 0.1 (9).

$$\boldsymbol{u}_{j,i,G} = \begin{cases} \boldsymbol{v}_{j,i,G} & \text{if } U[0,1] \leq CR_i \text{ or } j = j_{rand} \\ \boldsymbol{x}_{j,i,G} & \text{otherwise} \end{cases} \quad (8)$$

Where j_{rand} is randomly selected index of a feature, which has to be updated ($j_{rand} = U[1, D]$), D is the dimensionality of the problem.

$$CR_i = N\left[M_{CR,r}, 0.1\right] \quad (9)$$

Vector which will be in the next generation G+1 is selected by elitism. When the objective function value of trial vector $\boldsymbol{u}_{i,G}$ is better than that of the original vector $\boldsymbol{x}_{i,G}$, the trial vector will be selected for the next population and the original will be placed into the external archive A. Otherwise, the original will survive and the content of A remains unchanged (10).

$$\boldsymbol{x}_{i,G+1} = \begin{cases} \boldsymbol{u}_{i,G} & \text{if } f\left(\boldsymbol{u}_{i,G}\right) < f\left(\boldsymbol{x}_{i,G}\right) \\ \boldsymbol{x}_{i,G} & \text{otherwise} \end{cases} \quad (10)$$

EXPERIMENT SETTING

The regression capabilities of proposed version of the AP algorithm were tested on four typical benchmark functions used in (Koza 1994):

- Quintic: $y = x^5 - 2x^3 + x$
- Sextic: $y = x^6 - 2x^4 + x^3$
- 3Sine: $y = \sin x + \sin 2x + \sin 3x$
- 4Sine: $y = \sin x + \sin 2x + \sin 3x + \sin 4x$

The dataset of 50 points for Quintic and Sextic functions was generated in x range <-1, 1>. Also 50 points from x range <-π, π> were generated for function 3Sine and 4Sine. All four datasets are displayed in Figures 1, 2, 3 and 4.

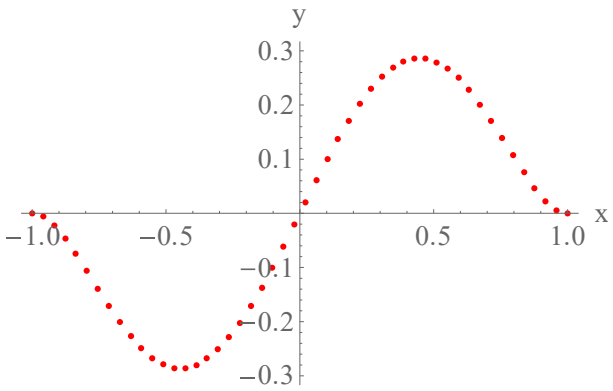

Figure 1: Quintic Dataset

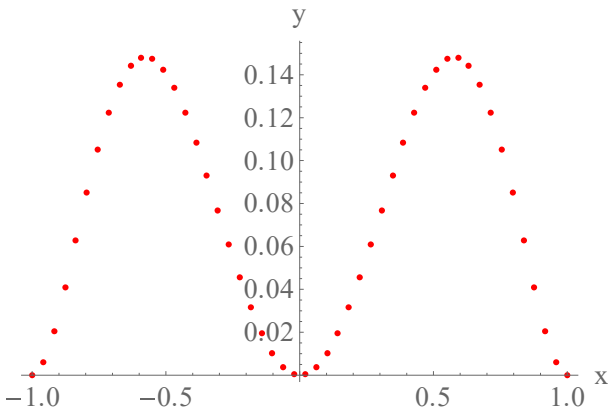

Figure 2: Sextic Dataset

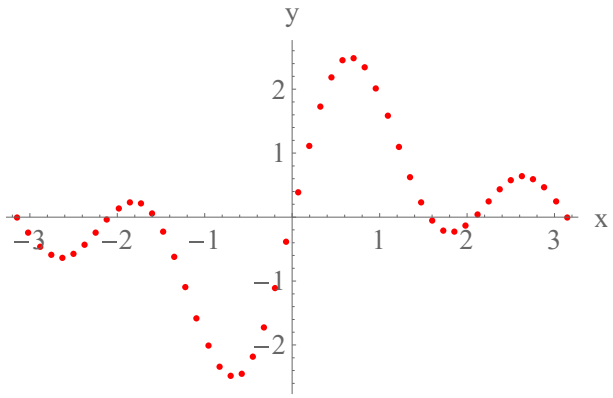

Figure 3: 3Sine Dataset

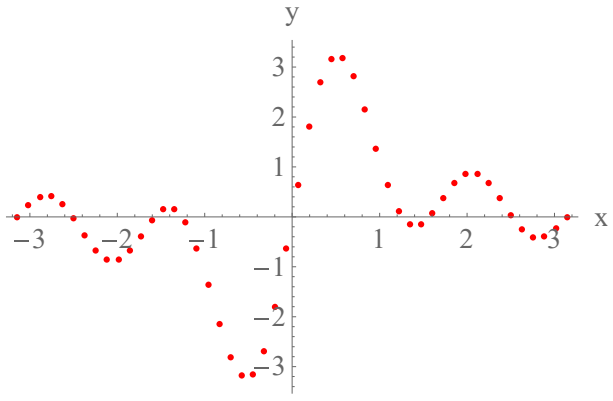

Figure 4: 4Sine Dataset

The settings of AP for the experiments and the settings of the underlying EA – SHADE can be found in Table 1.

Table 1: AP and SHADE Settings

	Quintic, Sextic	**3Sine, 4Sine**
AP		
GFS	+, -, *, /, *x*, *k*	
SHADE		
Dimension	90 (60 program + 30 constants)	180 (120 program + 60 constants)
Population size	50	100
Generations	4000	2000
Historical memory size	50	100

All datasets were synthesized 30 times and the results are presented in the next section.

RESULTS

Simple descriptive statistic results over 30 independent runs on all four datasets are given in Table 2. The table shows minimum, maximum, mean, median and standard deviation values obtained throughout the test. The fitness value of a synthesized program is a sum of the absolute distances between dataset points and their synthesized equivalents. Best results are displayed in Figures 5, 6, 7 and 8. The dataset is illustrated by red points and the best solution is represented by a green line.

Table 2: Results Over 30 Independent Runs of AP with SHADE and Extended Individuals

Dataset	**Min**	**Max**	**Mean**	**Med**	**StD**
Quintic	0.21	6.36	1.60	0.61	2.19
Sextic	0.24	2.37	0.75	0.62	0.54
3Sine	11.96	37.47	14.94	12.79	6.18
4Sine	14.23	26.09	16.54	15.77	2.57

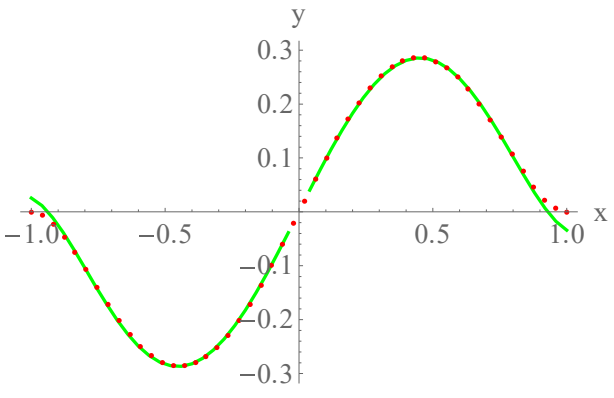

Figure 5: Quintic Regression

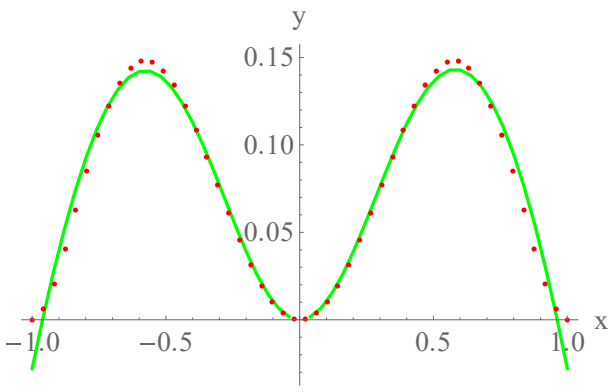

Figure 6: Sextic Regression

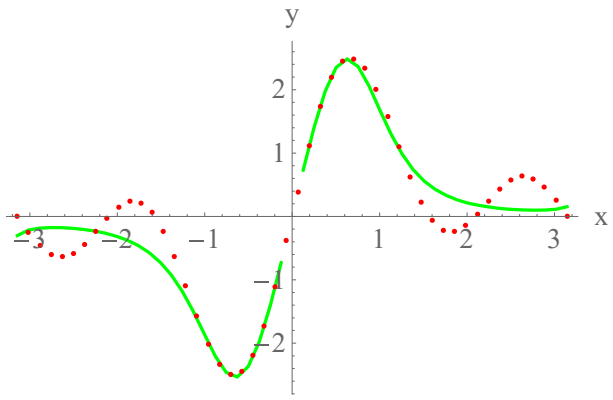

Figure 7: 3Sine Regression

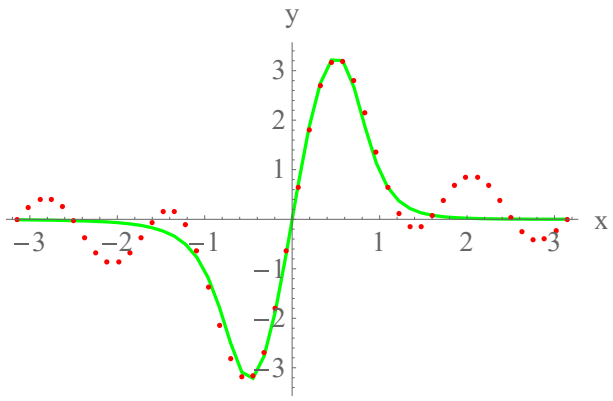

Figure 8: 4Sine Regression

As expected from the results in Table 2, the synthesized functions for 3Sine and 4Sine dataset are far less precise in modeling, than that of Quintic and Sextic datasets.
In order to obtain a better model for 3Sine and 4Sine datasets, the content of GFS was extended by *sin* function and the most precise models are shown in Figure 9 and Figure 10.

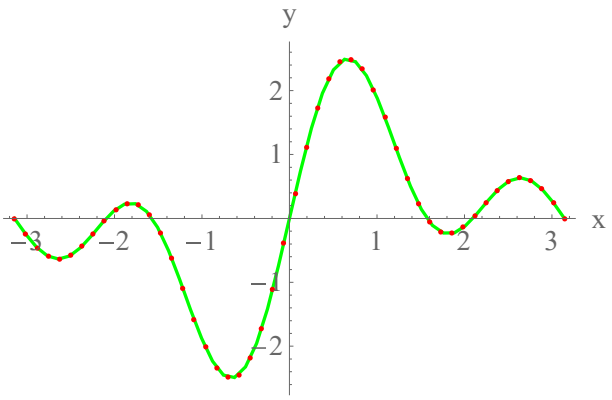

Figure 9: 3Sine Regression with *sin* in GFS

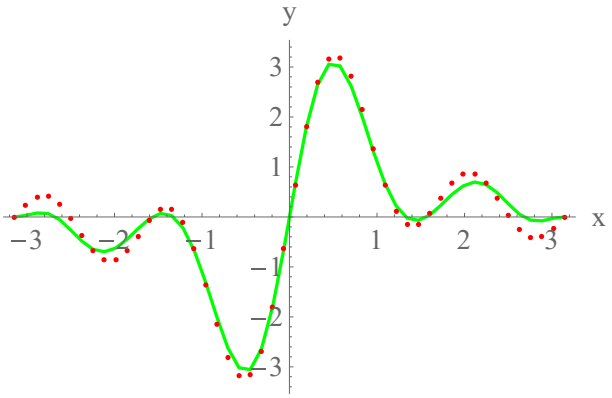

Figure 10: 4Sine Regression with *sin* in GFS

CONCLUSION

This paper presented a novel approach to estimation of the constant values in programs synthesized by AP. This approach uses part of the features of an individual in EA for the evolution of constant values.

As can be seen from the results, the performance of this variant is sufficient for two of the four benchmark test functions – Quintic and Sextic. Those two datasets are generated by polynomials and therefore can be synthesized from the functions and terminals contained in the GFS. Contrarily, 3Sine and 4Sine datasets are generated by harmonic functions which are not present in the GFS. This leads to the less precise regression, but the main trend is still transcribed. Further experiments confirmed, that when the harmonic function *sin* is added into the GFS, AP is able to find a better solution. This opens an interesting topic for further research in the possibility of using AP variant with extended individuals for the prediction of a suitable content of GFS. Determining the most suitable content of GFS for real world problems is a complex task and therefore a technique for prediction might be beneficial.

All four datasets tested in this study were taken from (Koza 1994). The proposed variant uses evolution for estimating the constant values, creating an infinite space of possibly synthesized programs and thus making it less suitable for non-complex test functions where the static value constants would be more reasonable (Quintic, Sextic, 3Sine, 4Sine, e. g.). The comparison between GP and AP on those datasets can be found in (Zelinka, Oplatkova 2003). One of the goals of this paper was to demonstrate that AP with extended individuals is still able to perform a regression which transcribes the original trend of a non-complex dataset, but the main area of use is for the regression of complex datasets where the continuous constant space is necessary.

The advantage of the proposed approach is in its time complexity which is increased against canonical AP only by the increase in size of an individual in used EA. While other approaches (meta-evolution and non-linear fitting) might provide more precise results, their time complexity is higher and a lot harder to estimate which might be a problem in real time regression and prediction. The time complexity of meta-evolution is dependent on settings of the secondary EA and the time complexity of the non-linear fitting approach strongly depends on the used fitting algorithm and on the number of constants to estimate.

Further research will be targeted at the use of proposed variant of AP in real world applications where the constant estimation will be more likely to bring benefits in regression and at improving its precision.

ACKNOWLEDGEMENT

This work was supported by Grant Agency of the Czech Republic – GACR P103/15/06700S, further by the Ministry of Education, Youth and Sports of the Czech Republic within the National Sustainability Programme Project no. LO1303 (MSMT-7778/2014). Also by the European Regional Development Fund under the Project CEBIA-Tech no. CZ.1.05/2.1.00/03.0089 and by Internal Grant Agency of Tomas Bata University under the Projects no. IGA/CebiaTech/2016/007.

REFERENCES

Koza, J. R. 1990. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Stanford University, Department of Computer Science.

Koza, J. R. 1994. Genetic programming II: Automatic discovery of reusable subprograms. Cambridge, MA, USA.

Oplatková, Z. and Zelinka, I. 2006. Investigation on artificial ant using analytic programming. In Proceedings of the 8th annual conference on Genetic and evolutionary computation (pp. 949-950). ACM.

Senkerik, R.; Oplatková, Z.; Zelinka, I. and Davendra, D. 2013. Synthesis of feedback controller for three selected chaotic systems by means of evolutionary techniques: Analytic programming. Mathematical and Computer Modelling, 57(1), 57-67.

Tanabe, R. and Fukunaga, A. 2013. Success-history based parameter adaptation for differential evolution. In Evolutionary Computation (CEC), 2013 IEEE Congress on (pp. 71-78). IEEE.

Zelinka, I. 2001. Analytic programming by means of new evolutionary algorithms, Proceedings of 1st International Conference on New Trends in Physics'01, Brno, Czech Republic, pp. 210-214.

Zelinka, I. and Oplatkova, Z. 2003. Analytic programming – comparative study, Proceedings of Second International

Conference on Computational Intelligence, Robotics, and Autonomous Systems, Singapore.

Zelinka, I.; Chen, G. and Celikovsky, S. 2008. Chaos synthesis by means of evolutionary algorithms. International Journal of Bifurcation and Chaos, 18(04), 911-942.

Zelinka, I.; Oplatkova, Z. and Nolle, L. 2005. Analytic programming–Symbolic regression by means of arbitrary evolutionary algorithms. Int. J. of Simulation, Systems, Science and Technology, 6(9), 44-56.

AUTHOR BIOGRAPHIES

ADAM VIKTORIN was born in the Czech Republic, and went to the Faculty of Applied Informatics at Tomas Bata University in Zlín, where he studied Computer and Communication Systems and obtained his MSc degree in 2015. He is studying his Ph.D. at the same university and the field of his studies are: Artificial intelligence, data mining and evolutionary algorithms. His email address is: aviktorin@fai.utb.cz

MICHAL PLUHACEK was born in the Czech Republic, and went to the Faculty of Applied Informatics at Tomas Bata University in Zlín, where he studied Information Technologies and obtained his MSc degree in 2011 and Ph.D. in 2016 with the dissertation topic: Modern method of development and modifications of evolutionary computational techniques. He now works as a researcher at the same university. His email address is: pluhacek@fai.utb.cz

ZUZANA KOMINKOVA OPLATKOVA is an associate professor at Tomas Bata University in Zlin. Her research interests include artificial intelligence, soft computing, evolutionary techniques, symbolic regression, neural networks. She is an author of around 100 papers in journals, book chapters and conference proceedings. Her email address is: oplatkova@fai.utb.cz

ROMAN SENKERIK was born in the Czech Republic, and went to the Tomas Bata University in Zlin, where he studied Technical Cybernetics and obtained his MSc degree in 2004, Ph.D. degree in Technical Cybernetics in 2008 and Assoc. prof. in 2013 (Informatics). He is now an Assoc. prof. at the same university (research and courses in: Evolutionary Computation, Applied Informatics, Cryptology, Artificial Intelligence, Mathematical Informatics). His email address is: senkerik@fai.utb.cz