Mutation and Crossover with Abstract Expression Grammars

Michael F. Korns
Freeman Investment Management
1 Plum Hollow
Henderson, Nevada 89052
1 (702) 837 3498

mkorns@korns.com

ABSTRACT

Simple enhancements to the standard population operators of mutation and crossover, utilizing Abstract Expression Grammars, are investigated. In previous works, Abstract Expression Grammars have been used to integrate Genetic Algorithms, Genetic Programming, Swarm Intelligence, and Differential Evolution, into a seamlessly unified approach to symbolic regression. In this work, the potential for Abstract Expression Grammars to have a direct impact on the classic Genetic Programming mutation and crossover operators is demonstrated. The features of abstract expression grammars are explored, details of abstract mutation and crossover are provided, and the beneficial effects of abstract mutation and crossover are tested with several published nonlinear regression problems.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming - Program Synthesis.

General Terms

terms: Algorithms

Keywords

keywords: Abstract Expression Grammars, Differential Evolution, Genetic Programming, Particle Swarm, Symbolic Regression.

1. INTRODUCTION

Large scale general nonlinear regression is currently practiced largely using genetic programming under the name Symbolic Regression [5]. However, as a nonlinear regression algorithm, genetic programming has a number of well known difficulties including expression bloat and lack of fine grain control over the solution produced. On the other hand, swarm intelligence [1] and differential evolution [8] provide excellent fine grain control but are not easily linked to general algebraic expressions.

In [3] and [4] Abstract Expression Grammars are used to combine swarm intelligence, differential evolution, and genetic programming into a seamlessly unified algorithm for general nonlinear regression. In this work, abstract expression grammars are used to directly enhance the classic genetic programming population operators of mutation and crossover into two new population operators: abstract mutation and abstract crossover.

The compelling rationale for abstract mutation and crossover will be developed before presenting a general overview of the processes of abstract mutation and abstract crossover. Background information on nonlinear regression systems and abstract expression grammars will be presented before developing the technical details of the processes of abstract mutation and abstract crossover. Finally, the beneficial effects of abstract mutation and abstract crossover, as compared with standard mutation and crossover, will be investigated by comparative testing on several published nonlinear regression problems.

2. Abstract Mutation and Crossover *Intro*

In genetic programming [5] the basic mutation population operator selects a random segment from an expression such as:

2.1 f = (log(x3)/**sin(x2*45.3)**)>x4 ? tan(x6) : cos(x3)

The selected segment is highlighted "**sin(x2*45.3)**" above. In standard mutation the selected segment is replaced with a new randomly constructed segment such as:

2.2 f = (log(x3)/**(2.6+log(x2))**)>x4 ? tan(x6) : cos(x3)

Notice that in standard mutation the selected segment "**sin(x2*45.3)**" may have nothing in common with the mutated segment "**(x5+log(2.6))**" which replaces it. In abstract mutation the selected segment "**sin(x2*45.3)**" is replaced with a related abstracted segment *abstract*["**sin(x2*45.3)**"]. Similarly, in genetic programming [5] the basic crossover population operator selects two random segments from two expressions such as:

2.3 f = (**log(x3)**/sin(x2*45.3))>x4 ? tan(x6) : cos(x3)

2.4 f = (log(x3)/cube(x2*45.3))>x4 ? **sin(x6)** : tan(x3)

The selected segments are swapped "*crossed over*" in the respective expressions as follows:

2.5 f = (**sin(x6)**/sin(x2*45.3))>x4 ? tan(x6) : cos(x3)

2.6 f = (log(x3)/cube(x2*45.3))>x4 ? **log(x3)** : tan(x3)

In abstract crossover each selected segment is abstracted before crossover *abstract*["**log(x3)**"] and *abstract*["**sin(x6)**"]. Before we can explain why the abstraction process is so logically compelling and what the actual details of the abstraction process are, we must provide some background information on nonlinear regression and on abstract expression grammars.

3. Background on Regression Systems

A general linear regression system accepts an input matrix, X, of N rows and M columns and a dependent variable vector, Y, of length N. The dependent vector Y is related to X thru the (hopefully but not necessarily linear) function, f, as follows: Y[n] = f(X[n]). The output of a linear regression system will be a coefficient vector, C, of length M, such that the inner product of C with each row of X produces an estimate vector, EY, which minimizes the least square error between EY and Y. General linear regression systems can easily be constructed using Gaussian methods.

For our purposes in the remainder of this paper, we will normalize all least squared error measurements by dividing by the standard deviation of Y. We will call this the Normalized Least Squared Error (NLSE).

A general nonlinear regression system accepts an input matrix, X, of N rows and M columns and a dependent variable vector, Y, of length N. The dependent vector Y is related to X thru the (quite possibly nonlinear) function, f, as follows: Y[n] = f(X[n]). The output of a nonlinear regression system will be a program object (which we will hereinafter call an Estimator Agent), f, such that invoking f on each row of X produces an estimate vector, EY, which minimizes the normalized least square error between EY and Y. General nonlinear regression systems can be constructed using genetic programming methods[3] [4] [5] [7].

The capabilities of the agent, f, will be constrained by some sort of algebraic expression grammar built into the nonlinear regression system. More to the point, the computer codes executed upon invoking f will be a compilation of some statement in said grammar.

4. A Concrete Expression Grammar

A simple concrete expression grammar suitable for use in most nonlinear regression systems would be a C-like grammar with the following basic elements.

4.1 Real Numbers: 3.45, -.0982, and 100.389

4.2 Row Features: x1, x2, and x5.

4.3 Operators: +, *, /, %, <, <=, ==, !=, >=, >

4.4 Functions: sqrt(), log(), cube(), sin(), tan(), max(), etc.

4.5 Conditional: (expr1 < expr2) ? expr3 : expr4

A nonlinear regression system might create its final estimator agent using mutation, cross over, or any number of techniques; but, the final estimator agent might easily be a compilation of a basic concrete expression such as:

4.6 f = (log(x3)/sin(x2*45.3))>x4 ? tan(x6) : cos(x3)

Computing an NLSE score for f requires only a single pass over every row of X and results in an attribute being added to f by executing the "score" method compiled into f as follows.

4.7 f.NLSE = f.score(X,Y).

5. Abstract Constants

Suppose that we are satisfied with the form of the expression in (4.6); but, we are not sure that the real constant 45.3 is optimal. The standard genetic programming algorithm does not provide a mechanism for optimizing the real constant, 45.3, other than running the symbolic regression system for more iterations; and, then we are not guaranteed of receiving an improved answer in the same form as in (4.6).

We can enhance our nonlinear regression system with the ability to optimize individual real constants by adding abstract constant rules to our built-in algebraic expression grammar.

5.1 Abstract Constants: c1, c2, and c10

Abstract constants represent placeholders for real numbers which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (4.6) as follows.

5.2 f = (log(x3)/sin(x2***c1**))>x4 ? tan(x6) : cos(x3)

The compiler adds a new real number vector, **C**, attribute to f such that f.C has as many elements as there are abstract constants in (5.2). Optimizing this version of f requires that the built-in "score" method compiled into f be changed from a single pass to a multiple pass algorithm in which the real number values in the abstract constant vector, f.C, are iterated until the expression in (5.2) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract constant vector, f.C, to optimal real number choices. Clearly the particle swarm [1] and differential evolution [8] algorithms provide excellent candidate algorithms for optimizing f.C and they can easily be compiled into f.score by common compilation techniques currently in the main stream.

Summarizing, we have a new grammar term, c1, which is a reference to the 1st element of the real number vector, f.C (*in C language syntax c1 == f.C[1]*). The f.C vector is optimized by scoring f, then altering the values in f.C, then repeating the process iteratively until an optimum NLSE is achieved.

Two important features of abstract expression grammars are worth mention here. The overall genetic programming algorithms within the nonlinear regression system do not have to be altered because the swarm and differential learning enhancements are hidden inside the "score" method by the abstract expression compiler when appropriate. Furthermore, as Riccardo Poli [11] has pointed out, a new population operator can be defined which converts abstract expressions into their concrete counterparts. For instance, the estimator agent in (5.2) is optimized with:

5.3 f.C == < 45.396 >

Then the optimized estimator agent in (5.2) has a concrete conversion counterpart as follows:

5.4 f = (log(x3)/sin(x2***45.396**))>x4 ? tan(x6) : cos(x3)

Since abstract expressions are not grammatically excessively different than concrete expressions, the genetic programming logic in the nonlinear regression system will operate on either type of expression. At different stages in the evolutionary process population operators can be introduced which convert abstract expressions into their optimized concrete counterparts, or even new mutation operators which convert concrete expressions into abstract expressions.

6. Abstract Features

Suppose that we are satisfied with the form of the expression in (4.6); but, we are not sure that the features, x2, x3, and x6, are optimal choices. The standard genetic programming algorithm does not provide a mechanism for optimizing these features other than running the symbolic regression system for more iterations; and, then we are not guaranteed of receiving an improved answer in the same form as in (4.6).

We can enhance our nonlinear regression system with the ability to optimize individual features by adding abstract feature rules to our built-in algebraic expression grammar.

6.1 Abstract Features: v1, v2, and v10

Abstract features represent placeholders for features which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (4.6) as follows.

6.2 f = (log(**v1**)/sin(**v2***45.3))>**v3** ? tan(**v4**) : cos(**v1**)

The compiler adds a new integer vector, **V**, attribute to f such that f.V has as many elements as there are abstract features in (6.2). Each integer element in the f.V vector is constrained between 1 and M, and represents a choice of feature (*in x*). Optimizing this version of f requires that the built-in "score" method compiled into f be changed from a single pass to a multiple pass algorithm in which the integer values in the abstract feature vector, f.V, are iterated until the expression in (6.2) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract feature vector, f.V, to integer choices selecting optimal features (*in x*). Clearly the genetic algorithm [6], discrete particle swarm [1], and discrete differential evolution [8] algorithms provide excellent candidate algorithms for optimizing f.V and they can easily be compiled into f.score by common compilation techniques currently in the main stream.

Summarizing, we have a new grammar term, v1, which is an indirect feature reference thru to the 1st element of the integer vector, f.V (*in C language syntax v1 == x[f.V[1]]*). The f.V vector is optimized by scoring f, then altering the values in f.V, then repeating the process iteratively until an optimum NLSE is achieved.

For instance, the estimator agent in (6.2) is optimized with:

6.3 f.V == < 2, 4, 1, 6 >

Then the optimized estimator agent in (6.2) has a concrete conversion counterpart as follows:

6.4 f = (log(**x2**)/sin(**x4***45.396))>**x1** ? tan(**x6**) : cos(x2)

7. Abstract Functions

Similarly, we can enhance our nonlinear regression system with the ability to optimize individual features by adding abstract functions rules to our built-in algebraic expression grammar.

7.1 Abstract Functions: f1, f2, and f10

Abstract functions represent placeholders for built-in functions which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (4.6) as follows.

7.2 f = (**f1**(x3)/**f2**(x2*45.3))>x4 ? **f3**(x6) : **f4**(x3)

The compiler adds a new integer vector, **F**, attribute to f such that f.F has as many elements as there are abstract features in (7.2). Each integer element in the f.F vector is constrained between 1 and (*number of built-in functions available in the expression grammar*), and represents a choice of built-in function. Optimizing this version of f requires that the built-in "score" method compiled into f be changed from a single pass to a multiple pass algorithm in which the integer values in the abstract function vector, f.F, are iterated until the expression in (7.2) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract function vector, f.F, to integer choices selecting optimal built-in functions. Clearly the genetic algorithm [6], discrete particle swarm [1], and discrete differential evolution [8] algorithms provide excellent candidate algorithms for optimizing f.F and they can easily be compiled into f.score by common compilation techniques currently in the main stream.

Summarizing, we have a new grammar term, f1, which is an indirect function reference thru to the 1st element of the integer vector, f.F (*in C language syntax f1 == funtionList[f.F[1]]*). The f.F vector is optimized by scoring f, then altering the values in f.F, then repeating the process iteratively until an optimum NLSE is achieved.

For instance, if the valid function list in the expression grammar is

7.3 < log, sin, cos, tan, max, min, avg, cube, sqrt >

And the estimator agent in (7.2) is optimized with:

7.4 f.F == < 1, 8, 2, 4 >

Then the optimized estimator agent in (7.2) has a concrete conversion counterpart as follows:

7.5 f = (**log**(x3)/**cube**(x2*45.3))>x4 ? **sin**(x6) : **tan**(x3)

The built-in function argument arity issue is easily resolved by having each built-in function ignore any excess arguments and substitute defaults for any missing arguments or by having the number of arguments automatically restrict the choice of concrete functions to those with the proper arity.

Furthermore random noise functions, such as in [9], can easily be added to the list of available built-in functions in the expression grammar.

8. Abstract Mutation and Crossover *Details*

A set of simple rules define the process of abstracting an expression segment in both abstract mutation and abstract crossover as follows:

8.1 Real Numbers: 3.45, -.0982 ➔ c0, c1

8.2 Row Features: x1, x4 ➔ v0, v1

8.3 Operators: +, *, / ➔ f0, f1

8.4 Functions: sqrt(), log(), cube() ➔ f0, f1, f3

Using these simple rules, the abstract mutation population operator selects a random segment from an expression such as:

8.5 f = (log(x3)/**sin(x2*45.3)**)>x4 ? tan(x6) : cos(x3)

The selected segment is highlighted "**sin(x2*45.3)**" above. In abstract mutation the selected segment is replaced with its abstract conversion where sin ➔ f0, x2 ➔ v0, * ➔ f1, and 45.3 ➔ c0 as follows:

8.6 f = (log(x3)/**f0(f1(v0,c0))**)>x4 ? tan(x6) : cos(x3)

Similarly, the abstract crossover population operator selects two random segments from two expressions such as:

8.7 f = (**log(x3)**/sin(x2*45.3))

8.8 f = (tan(x3)/cube(**x2*45.3**))

The selected segments are first "*abstracted*" and then swapped "*crossed over*" in the respective expressions as follows:

8.9 f = (**(x2*45.3)**/sin(x2*45.3))

which is abstracted into:

8.10 f = (**(f0(v0,c0))**/sin(x2*45.3))

8.11 f = (tan(x3)/cube(**log(x3)**))

which is abstracted into:

8.12 f = (tan(x3)/cube(**f0(v0)**))

After abstract mutation or crossover, the new abstract expressions are optimized by the regression system and their ***optimized concrete conversions*** are saved in proper the evolutionary populations.

The compelling first principles argument for abstract mutation and crossover is as follows. It is likely that the regression system will find optimized substitutions for **f0**, **v0**, and **c0** which will make

8.10 f = (**(f0(v0,c0))**/sin(x2*45.3))

a more fit individual than the simple

8.9 f = (**(x2*45.3)**/sin(x2*45.3))

Especially since (8.9) was created by swapping "**(x2*45.3)**" into (8.7) without any analysis of the consequences.

Of course a compelling first principles argument is no substitute for experimental evidence. So the remainder of this work details the results from comparative testing standard mutation and crossover with abstract mutation and crossover on several difficult previously problems.

9. Testing Regimen

We use the nine base test cases from [3] as a training set, to test for improvements in accuracy. Our testing regimen uses only statistical best practices out-of-sample testing techniques. We test each of the nine test cases on matrices of 10000 rows by 5 columns with no noise, and on matrices of 10000 rows by 20 columns with 40% noise, before drawing any performance conclusions. Taking all these combinations together, this creates a total of 18 separate test cases.

For each test a training matrix is filled with random numbers between -50 and +50. The target expression for the test case is applied to the training matrix to compute the dependent variable and the required noise is added.

The symbolic regression system is trained on the training matrix to produce the regression champion. Following training, a testing matrix is filled with random numbers between -50 and +50. The target expression for the test case is applied to the testing matrix to compute the dependent variable and the required noise is added. The regression champion is evaluated on the testing matrix for all scoring (i.e. out of sample testing).

Standard regression studies often utilize least squares error (LSE) as a fitness measure. In our case we normalize by dividing LSE by the standard deviation of "Y" (dependent variable). This normalization allows us to meaningfully compare the normalized least squared error (NLSE) between different problems.

10. Previous Results on Nine Base Problems

The previously published results [4] of training on the nine base training models on 10,000 rows and five columns training for 20 generations with no random noise and *no abstract mutation or crossover*, are shown below (The nine base test cases are described in detail in [3]).

In general, training time is very reasonable given the difficulty of some of the problems and the limited number of training generations. Average error varies from excellent to poor with the *linear* and *cubic* problems showing the best performance.

In some of the test cases, testing error is either close to or exceeds the standard deviation of Y (not very good); however, in many of the test cases classification is below 0.20. (very good).

Unfortunately, extreme differences between training error and testing error in the very difficult *mixed* and *ratio* problems suggest over-fitting and bring into question to relevance of of the low training errors in those test cases.

Table 10.1: Results for 10K rows by 5 cols no random noise

Test Case	Train NLSE	Test NLSE
Linear	.01	.01
Cubic	.00	.00
hidden	.00	.05
Cross	.37	.39
elipse	.00	.00
cyclic	.04	.14
hyper	.00	.00
mixed	.24	1.65
ratio	.03	1.05

The previously published results [4] of training on the nine base training models on 10,000 rows and twenty columns training for 20 generations with 40% random noise and *no abstract mutation or crossover*, are shown below.

Table 20.2: Results for 10K rows by 20 cols 40% random noise

Test Case	Train NLSE	Test NLSE
Linear	.11	.11
Cubic	.11	.11
hidden	.99	.99
Cross	.80	.80
elipse	.45	.46
cyclic	.39	.91
hyper	.96	.96
mixed	.69	1.85
ratio	.95	1.18

Clearly the system, without abstract mutation and crossover, performs most poorly on the test cases *hidden*, *mixed* and *ratio* with conditional target expressions. There is much evidence of over-fitting shown by the extreme differences between training error and testing error on some test cases.

11. Enhanced Results on Nine Base Problems

The enhanced results on the nine base training models on 10,000 rows and five columns training for 20 generations with no random noise but *with abstract mutation and crossover*, are shown below.

Table 31.1: Results for 10K rows by 5 cols no random noise

Test Case	Train NLSE	Test NLSE
Linear	.00	.00
Cubic	.00	.00
hidden	.00	.00
Cross	.00	.00
elipse	.00	.00
cyclic	.02	.00
hyper	.00	.00
mixed	.97	.98
ratio	.96	.98

The enhanced results on the nine base training models on 10,000 rows and twenty columns training for 20 generations with 40% random noise and *with abstract mutation and crossover*, are shown below.

Table 41.2: Results for 10K rows by 20 cols 40% random noise

Test Case	Train NLSE	Test NLSE
Linear	.11	.11
Cubic	.11	.11
hidden	.11	.11
Cross	.69	.72
elipse	.42	.43
cyclic	.39	.35
hyper	.48	.50
mixed	.92	.96
ratio	.91	.95

Clearly, adding abstract mutation and crossover has improved system performance. On most tests, performance is very reasonable given the difficulty of some of the problems and the limited training time allocated. Even on the more difficult *mixed* and *ratio* problems all evidence of over-fitting has disappeared. Normalized least squared error varies from excellent to poor with the *linear*, *cubic*, and *hidden* problems showing the best performance even with 40% noise.

These positive results have been achieved by simply converting 5% of all standard mutations to abstract mutations, and converting 5% of all standard crossovers to abstract crossovers. Empirical analysis of the mutation and crossover population operators shows that standard mutation and crossover tend to jump out of local minima into different random areas of the fitness landscape; while abstract mutation and crossover tend to provide *relatively* more fine grain exploration of the local landscape.

12. Training Epochs in Detail

The following test case provides an excellent example of how training proceeds faster with 5% abstract mutation and crossover than with only standard mutation and crossover.

12.1 squareRoot: y = 1.23 + (1.23*(sqrt(x1*x1*x1))
- (9.16*sqrt(x1*x2*x2))
+ (11.27*sqrt(x1*x2*x3))
+ (7.42*sqrt(x2*x3*x4))
+ (8.21*sqrt(x3*x4*x5)))

Using the nonlinear regression system described in [3] [4], we construct an X matrix 10,000 by 5, filled with random numbers between -50 and +50, and run the "squareRoot" training model (12.1) on each row of X to create the Y dependent vector. When we train quickly (*for only 50 generations*) we see the following interim training errors at the end of each epoch.

Table 52.2: Regression Results during 50 Generations

After Generation	Interim Training NLSE (standard only)	Interim Training NLSE (5% abstract)
10	**.9971**	**.9956**
20	**.9520**	**.7790**
30	**.8512**	**.5987**
40	**.7618**	**.4668**
50	**.6362**	**.2805**
Final	**.6312**	**.2803**

Clearly the training run converges faster with 5% abstract mutation and crossover than with only standard mutation and crossover.

13. Summary

There appears little benefit from keeping such valuable, yet disparate, algorithms as GA, GP, Particle Swarm, Differential Evolution, and even Gaussian Regression, separated, in isolation, and prevented from working together smoothly in a commercial nonlinear regression system. Abstract Expression Grammars have the potential to integrate Genetic Algorithms, Genetic Programming, Swarm Intelligence, Differential Evolution, and even Gaussian Regression into a seamlessly unified array of tools for use in industrial strength nonlinear regression systems.

Furthermore, in this work, abstract expression grammars provide value in directly enhancing the classic genetic programming population operators of mutation and crossover. The new population operators, abstract mutation and abstract crossover, provide both a compelling ex-ante rationale and compelling ex-post empirical evidence for the beneficial effects of abstract mutation and crossover, as compared with standard mutation and crossover, in comparative testing on several published nonlinear regression problems.

14. ACKNOWLEDGMENTS

Our thanks to Riccardo Poli for his thoughtful suggestions on new population operators possible with abstract expression grammars.

15. REFERENCES

[1] Eberhardt, Russel, Shi, Yuhui, and Kennedy, James. 2001 Swarm Intelligence. Morgan Kaufmann, New York, USA. http://www.amazon.com/Swarm-Intelligence-Morgan-Kaufmann-Artificial/dp/1558605959/ref=pd_bbs_sr_1?ie=UTF8&s=books&qid=1228938121&sr=8-1

[2] Hornby, Gregory S. 2006. ALPS: The Age-Layered Population Structure for Reducing the Problem of Premature Convergence. In Keijzer, Maarten, Catolico, Mike, Arnold, Dirk, Babobiv, Vladan, Blum, Christian, Bosman, Peter, Butz, Martin, V., Coello Coello, Carlos, Dasgupta, Dipankar, Ficici, Sevan, G., Foster, James, Hernandez-Aguirre, Arturo, Hornby, Greg, Lipson, Hod, McMinn, Phil, Moore, Jason, Raidl, Gunter, Rothlauf, Franz, Ryan, Conor, and Thierens, Dirk, editors, GECCO 2006: Proceedings of the 8th annual conference on Genetic and Evolutionary Computation, volume 1, pages 815-822, Seattle, Washington, USA. ACM Press. http://portal.acm.org/citation.cfm?id=1143997&coll=GUIDE&dl=GUIDE&CFID=14570833&CFTOKEN=82862158.

[3] Korns, Michael F. 2007. Large-Scale, Time-Constrained Symbolic Regression-Classification. In Riolo, Rick, L, Soule, Terrance, and Wortzel, Bill, editors, Genetic Programming Theory and Practive V, pages 53-68, New York, New York, USA. Springer. http://www.springer.com/computer/artificial/book/978-0-387-76307-1

[4] Korns, Michael F., and Nunez, Loryfel, 2008. Profiling Symbolic Regression-Classification. In Riolo, Rick, L, Soule, Terrance, and Wortzel, Bill, editors, Genetic Programming Theory and Practive VI, pages 215-228, New York, New York, USA. Springer. http://www.springer.com/computer/artificial/book/978-0-387-87622-1

[5] Koza, John, R. 1992 Genetic Programming: On Programming Computers by means of natural Selection. MIT Press, Cambridge, USA. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=5888

[6] Man, Kim-Fung, Tang, Kit-Sang, and Kwong, Sam. 1999. Genetic Algorithms. Springer, New York, USA. http://www.springer.com/engineering/robotics/book/978-1-85233-072-9

[7] O`Neil, Michael, and Ryan, Conor. 2003. Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Dortrecht, Netherlands. http://www.alibris.com/booksearch?binding=&mtype=&keyword=Grammatical+Evolution&hs.x=6&hs.y=15

[8] Price, Kenneth, Storn, Rainer, and Lampinen, Jouni 2005. Differential Evolution: A Practical Approach to Global Optimization. Springer, New York, USA. http://www.springer.com/computer/foundations/book/978-3-540-20950-8

[9] Schmidt, Michael D, and Lipson, Hod. 2007. Learning Noise. In Thierens, Dirk, Beyer, Hans-Georg, Bongard, Josh, Branke, Jurgen, Clark, John Andrew, Cliff, Dave, Congdon, Clare Bates, Deb, Kalyanmoy, Doerr, Benjamin, Kovacs, Tim, Kumar, Sanjeev, Miller, Julian F., Moore, Jason, Neumann, Frank, Pelikan, Martin, Poli, Riccardo, Sastry, Kumara, Stanley, Kenneth Owen, Stutzle, Thomas, Watson, Richard A., Wegener, Ingo, editors, GECCO 2007: Proceedings of the 9th annual conference on Genetic and Evolutionary Computation, volume 2, pages 1680-1685, London. ACM Press. http://portal.acm.org/citation.cfm?id=1143997&coll=GUIDE&dl=GUIDE&CFID=14570833&CFTOKEN=82862158.

[10] Cristianini, Nello, and Shawe-Taylor, John, 2000. An Introduction to Support Vector Machines and Other Kernel-based Learning Mmethods. Cambridge University press. http://www.amazon.com/Introduction-Support-Machines-Kernel-based-Learning/dp/0521780195/ref=pd_bbs_sr_2?ie=UTF8&s=books&qid=1228947802&sr=8-2

[11] Poli, Riccardo, Langdon, William B., and McPhee, Nicholas Freitag, 2008. A Field Guide to Genetic Programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (with contributions by J. R. Koza).