Fractional Genetic Programming for a More Gradual Evolution*

Artur Rataj

Institute of Theoretical and Applied Computer Science,
Bałtycka 5, Gliwice, Poland
arataj@iitis.gliwice.pl

Abstract. We propose a softening of a genetic program by the so–called fractional instructions. Thanks to their adjustable strengths, a new instruction can be gradually introduced to a program, and the other instructions may gradually adapt to the new member. In this way, a transformation of one candidate into another can be continuous. Such an approach makes it possible to take advantage of properties of real–coded genetic algorithms, but in the realm of genetic programming. We show, that the approach can be successfully applied to a precise generalisation of functions, including those exhibiting periodicity.

Keywords: genetic programming, real–coded genetic algorithm, evolutionary method, gradualism

1 Introduction

One of the basic concepts in genetics is quantitative inheritance, i.e. a gradual regulation of the strength of a single genetic trait. Such an inheritance is realised by e.g. encoding a single trait with several genes, which collaboratively add up to strengthen the trait.

A quantitative inheritance enables an easy way for an effective *exploitative* searching of the candidate space – an offspring, which is similar to its parents, has a high chance of being viable in similar environmental conditions, and thus to be a next step in walking the candidate space. The similarity might make it more difficult to find a substantially different environment by the offspring, yet a need to live in a radically different environment is often not the case. Consider e.g. the formation of limbs in animals – they evolved from fins in the process of a long, exploitative transition, that employed slight changes in expressions of genes and a gradual appearance of new genes. Actually, it has been recently shown, that just varying of the expression of certain genes in fish makes their fins more limb–like [5]. Such a graduality enabled a progressive change of the environment from sea to land, with children being able to live in an environment similar to that of their parents, though.

* This work has been supported under the project *An active creation of a model of space using a 3d scanner and an autonomous robot, N N516 440738*

The concept of graduality or continuity of the candidate space is the basis of real–coded genetic algorithms [6, 14], supporting a number of dedicated genetic operators [7].

We propose genetic programs in a form that is halfway between a tree representation [3] and a linear list [11]—a linear sequence of functions (instructions) is communicating using a stack, as introduced by [12], The basic difference of our approach, though, lies in the graduality of the stack operations, constructed to support the discussed continuity of instructions. The continuity is similar to that of [13]. There are two basic differences, though:

- No need for special continuous equivalents of common operations like addition or multiplication. Instead, the original functions can be used directly. The graduality is provided elsewhere—by a specially constructed stack.
- A program does not strictly follow a tree structure, as a value of a node can be reused by a number of other nodes, which promotes reusability of an instruction result and in effect reduces redundancy. Like in the case of instructions, the dependencies out of the tree hierarchy can also form gradually.

The paper is constructed as follows: Sect. 2 describes fractional stack operations. On the basis of these definitions, fractional operations are discussed in Sect. 3. These build a fractional program, described in Sect. 4. A method of evolving such programs is introduced in Sect. 5. Section 6 presents some tests. Finally, there is a discussion in Sect. 7.

2 A Fractional Stack

Let the stack, in order to be compatible with the fractional instructions, have a continuous length, so that it can support a fractional pushing and popping. Thanks to this, a weak operation, i.e. having a low strength, may interact with the stack so that there is only a minimal change to the stack state.

Let each element on the stack have a real value x and also a real length. The length is specified by the strength s of a push operation $\mathrm{push}(s, x)$, and contributes to the total length of the stack, which is a different value than the total number of elements in that stack, i.e. the stack size. A pop operation, in turn, $y = \mathrm{pop}(s)$, having the strength s, shortens the stack length by s, by removing 0 or more top elements from the stack, and also possibly shortening the length of the element f, which is on the top after that removal. The value popped y is a mean of values of the elements removed or shortened, weighted by the respective lengths of each element removed, and also by the amount of shortening of the length of f.

Let us consider an example. Two operations $\mathrm{push}(0.6, 10)$ and $\mathrm{push}(1.2, 20)$ led to a stack whose top fragment is shown in Fig. 1(a). Then, a $\mathrm{pop}(1.5)$ operation, in order to shorten the stack by a proper value, needed to remove the top element and shorten the neighbouring element by 0.3, as seen in Fig. 1(b). The

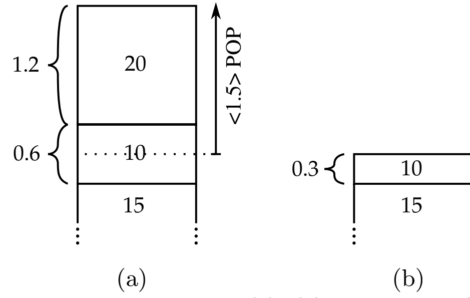


Fig. 1. An example of a fractional pop(s): (a) before, and (b) after the operation.

value returned by that operation would be a weighted average

$$\frac{1.2 \cdot 20 + 0.3 \cdot 10}{1.2 + 0.3} = 18.$$

Let i be an index of an element in the stack, 0 for the bottommost element, $S - 1$ for the topmost element, where S is the total number of elements in the stack. Then let L_i be the length of the ith element in the stack.

3 Fractional Instructions

Given the stack operations, definitions of the fractional instructions are trivial. Possible arguments of such an instruction are always popped from the stack, and a possible result of the instruction is pushed back to the stack. A length of each of the stack operations is given by the instruction *strength*. A low–strength instruction can only minimally modify the stack contents, fulfilling in this way the graduality criterion. In particular, a zero–strength instruction does nothing.

Let the set of instructions be limited to several stack handling operations, and also a handful of arithmetic operations. Each instruction's mnemonic is preceded by the instruction's strength. Table 3 lists the definitions of all instructions. Arguments of binary operators are evaluated from left to right. As seen, $\langle s \rangle$ POP does not use the popped value, the whole role of the instruction is to modify the stack state. We introduce $\langle s \rangle$ COPY n:x, that multiplies the length of n elements at the stack top by x. The instruction is modelled after the instruction DUP – an equivalent of COPY 1:2, used to simplify programs implemented in stack–based languages, like FORTH, FIFTH [8] or Java bytecode [10]. In our case of fractional instructions, COPY can act as an adapter of n argument passed between operations of different strengths.

4 A Fractional Program

Before the program is executed, its N_i input values X_i, $i = 0, 1, \ldots N_i - 1$, are pushed to the stack with a sequence of operations $\langle 1 + p \rangle$ PUSH X_i. The fixed

Table 1. A list of basic instructions.

Mnemonic	Stack operations
$\langle s \rangle$ PUSH x	$\mathrm{push}(s, x)$
$\langle s \rangle$ POP	$\mathrm{pop}(s)$
$\langle s \rangle$ COPY $n{:}x$	$l_0 = L_{S-1}, v_0 = \mathrm{pop}(l_0)$, $l_1 = L_{S-1}, v_1 = \mathrm{pop}(l_1)$, $\vdots$ $l_{n-1} = L_{S-1}, v_{n-1} = \mathrm{pop}(l_{n-1})$, $\mathrm{push}(xl_{n-1}, v_{n-1})$, $\mathrm{push}(xl_{n-2}, v_{n-2})$, $\vdots$ $\mathrm{push}(xl_0, v_0)$
$\langle s \rangle$ ADD	$\mathrm{push}\Big(s, \mathrm{pop}(s) + \mathrm{pop}(s)\Big)$
$\langle s \rangle$ MUL	$\mathrm{push}\Big(s, \mathrm{pop}(s)\mathrm{pop}(s)\Big)$
$\langle s \rangle$ NEG	$\mathrm{push}\Big(s, -\mathrm{pop}(s)\Big)$
$\langle s \rangle$ INV	$\mathrm{push}\Big(s, \mathrm{pop}(s)^{-1}\Big)$
$\langle s \rangle$ SIN	$\mathrm{push}\Big(s, \sin\Big(\mathrm{pop}(s)\Big)\Big)$

value $p \geq 0$ is a padding, that decreases the probability of a stack underflow – the program may instead have a chance of popping the input value or its part again. This is the only case of an instruction, that has a strength greater than 1, but such an instruction never occurs in a program itself. We will use a padding of 1 in tests.

If it is unlikely that a program needs a very long stack to solve a given problem, the stack size might have some constant limit imposed, so that stack overflows become possible. This way, such programs, as likely too complex, are deemed invalid, which favours simpler solutions and also saves time by preventing a further exploitation of such programs.

A program, after finishing, yields a single output value Y, which is equal to whatever is left in the stack, but excluding its possible bottommost part, if any, never used by the program, as that part contains unprocessed input values. Thus,

$$Y = \mathrm{pop}\left(\left(\sum_{k=0}^{S-1} L_k\right) - \min\left(L_u, N_i\right)\right) \tag{1}$$

where L_u is the global minimum length of the stack across all $\mathrm{pop}(s)$ instructions executed within the program, or ∞ if there were no such instructions.

See that if the first instruction of a program is $\langle 1 \rangle$ COPY $N_i{:}z$, and then all of the following instructions are of a strength z, then for given X_i and any $z > 0$ the program yields exactly the same Y. This shows, that an instruction retains its characteristics for any strength, beside a different level of interaction with the

stack. What is important, though, is the ratio between strengths of instructions that communicate using the same stack, and not the absolute values of these strengths.

5 Evolutionary Method

We have developed an evolutionary method that dynamically adjusts the exploration/exploitation scheme like it is often practised in genetic programming [1, 4, 9]. The exploitation employs a novel technique that emphasises the advantage of gradual instructions. To keep this paper focused, we do not go into the complexities of maintaining a population of candidates.

5.1 Exploration – a new candidate

A candidate program has a fixed length P_n and a fixed maximum stack size P_s, both roughly accommodated to the function to fit to, so that too complex solutions are impossible.

There are several types of instructions to randomly pick. A priority is given to $\langle s \rangle$ PUSH x, as otherwise most candidates would turn out to be invalid due to stack underflows. We have chosen in tests that this type will be chosen with $P_{\text{PUSH}} = 0.3$, while for the 7 other types $P_{\neg\text{PUSH}} = 0.1$. The pushed value will be drawn using a uniform distribution, spanning reasonable limits $\langle -10, 10 \rangle$. Arguments of $\langle s \rangle$ COPY $n{:}x$ will be chosen using an uniform distribution as well. The ranges are $n \in \langle 1, 2 \rangle$ as it accommodates a typical number of values popped by an instruction; $x \in \langle 0, 2 \rangle$ not getting too large, as sequences of $\langle s \rangle$ COPY $n{:}x$ may effectively enlarge the upper limit of multiplying lengths of stack elements.

5.2 Exploitation – tuning of a candidate

A gradual exploitation shows the flexibility of the fractional programs. An instruction may have its strength gradually reduced even until it disappears, or another instruction may appear with its strength near 0. Both such a deletion and such an insertion may have an arbitrarily small influence on the program, as very small strengths translate to only a minimal interaction with the stack.

The exploitation is a random walk in the candidate space, beginning with a new candidate created by exploration. A backtrack to the previous position occurs, if the mean square error (MSE) of the new candidate generalised of fitting to a function $g(x)$ is not better than the respective MSE of the previous candidate, or if the new candidate is invalid because of a stack underflow or overflow, or because of an arithmetic error like a division by zero. Let a single step be called a modification Δ.

We are unsure which Δ is the best one for a given problem. Thus, we choose to introduce a variability of its parameters – they are picked using probability distributions before each step. Let these parameters be as follows – a minimum threshold $\Theta_{\min}$ and an instruction perturbance level Φ, both chosen from $\langle 0, 1 \rangle$

using an uniform distribution. $\Theta_{\min}$ controls how many instructions are modified on average within Δ. The idea is, that it would sometimes be enough to modify only a single instruction within Δ, and it might even be advantageous to do so, if exactly a single instruction needs to be modified, in order to get a better candidate; parallel modifications of other instructions might create an unwanted drift in the candidate space in such a case. Yet, sometimes more instructions need to be modified at once, in order to omit the Hamming wall [2]. Also, a computation time is obviously crucial in genetic programming, and the random walk might be faster at times if more instructions are modified per Δ. We allow each case by $\Theta_{\min}$ being variable. Φ stems from a similar reasoning – it translates to how much an instruction is modified on average within Δ. Sometimes a fast walk is needed when e.g. a candidate is far from the optimum point in the candidate space. Yet, sometimes small, precise steps are needed instead, when e.g. the exploited candidate oscillates very closely around the optimum point.

Let within a single Δ, each of the instructions I be modified as follows:

1. Pick $\Theta \in \langle 0, 1\rangle$ using an uniform distribution. If $\Theta < \Theta_{\min}$, then do not modify I. Go to 2 only otherwise.
2. Let a strength perturbance $\Phi_s = \delta_s \Phi$ and a value perturbance $\Phi_v = \delta_v \Phi$ control, respectively, how much an instruction strength and an instruction argument can be perturbed. We picked in tests fixed $\delta_s = \delta_v = 1/20$ as a trade–off between the exploitation speed and precision.
3. Let $\langle s\rangle$ be the current strength of I, and $\langle s\rangle'$ the modified one. Let
$$\langle s\rangle' = \max\left(0, \min\left(1, \langle s\rangle + \text{uni}\Big(-\frac{\Phi_s}{2}, \frac{\Phi_s}{2}\Big)\right)\right), \tag{2}$$
where $\text{uni}(a, b)$ picks a random value in the range $\langle a, b\rangle$ using an uniform distribution.
4. If $\langle s\rangle' = 0$, then let I be removed. Go to 5 only otherwise.
5. If I is of the type $\langle s\rangle$ PUSH v, let the pushed value v be modified to become v':
$$v' = \max\left(-10, \min\left(10, v + \text{uni}\Big(-\frac{\Phi_v}{2}, \frac{\Phi_v}{2}\Big)\right)\right). \tag{3}$$
6. If I is of the type $\langle s\rangle$ COPY $n{:}m$, let the length multiplier m be modified to become m':
$$m' = \max\left(0, \min\left(2, m + \text{uni}\Big(-\frac{\Phi_v}{2}, \frac{\Phi_v}{2}\Big)\right)\right). \tag{4}$$

Note that v and m are clipped to the same range as when a new instruction is created during the exploration.

After the instructions are modified, and if there have been $r > 0$ instructions removed during that process, then let r new instructions be picked, exactly as during the exploration, but let these new instructions be randomly inserted to the program, so that it retains the original number of instructions P_n. Let the strength of each be Φ_s, that is, a small value related to a maximum single modification of an instruction strength.

5.3 Adapting the exploration/exploitation ratio

Let the fitting quality of a candidate $\mathcal{C}$ to $g(x)$ be $\mathrm{MSE}(\mathcal{C})$. Let the best candidate so far be $\mathcal{B}$. If none, assume $\mathrm{MSE}(\mathcal{B}) = \infty$.

The *search loop* consists of two phases: explore by creating a new candidate $\mathcal{T}$, and then exploit $\mathcal{T}$, with the extent related to the ratio of quality of $\mathcal{T}$ to $\mathcal{B}$. If, after the whole exploitation stage, $\mathrm{MSE}(\mathcal{T}) < \mathrm{MSE}(\mathcal{B})$, then let $\mathcal{T}$ become the new $\mathcal{B}$. The loop ends, if $\mathrm{MSE}(\mathcal{B}) \leq \mathrm{MSE}_{\mathrm{max}}$, where $\mathrm{MSE}_{\mathrm{max}}$ is a fixed value, chosen depending on the minimum acceptable quality of the candidate which we want to find.

The mentioned extent decides on the exploration/exploitation balance, and also on the distribution of the computation time to the exploitations of candidates $\mathcal{T}$, given their relative quality. As $\mathrm{MSE}(\mathcal{T})$ may decrease during the exploitation, let the extent be dynamically updated using the current value of $\mathrm{MSE}(\mathcal{T})$. The extent is represented by a maximum failure count (MFC). The value expresses the maximum possible number of subsequent exploitation steps, which do not result in finding a better candidate. Thus, if a new path to improve $\mathcal{T}$ is found, then the path is given a chance, irrespective of the length of exploitation of the candidate so far. If MFC is reached, the exploitation is terminated, and the search loop returns to the exploration phase as described. MFC is computed for a new $\mathcal{T}$, and, to adapt it dynamically as discussed, also whenever the exploitation of $\mathcal{T}$ results in a candidate with lower $\mathrm{MSE}(\mathcal{T})$.

Let us consider a formula for MFC. Let its minimum value be $\mathrm{MFC}_{\mathrm{min}}$, so that any candidate is given a moderate chance to improve, irrespective of that candidate's quality. We want, though, to give a considerably greater MFC for candidates whose initial quality is estimated to be decent. Let the formula have a fixed parameter $t_r > 0$ to reflect that:

$$\mathrm{MFC} = \lfloor t_r \left(\tanh \left(t_s \frac{\mathrm{MSE}(\mathcal{T})}{\mathrm{MSE}(\mathcal{B})} - 1 \right) + 1 \right) + \mathrm{MFC}_{\mathrm{min}} + \frac{1}{2} \rfloor. \tag{5}$$

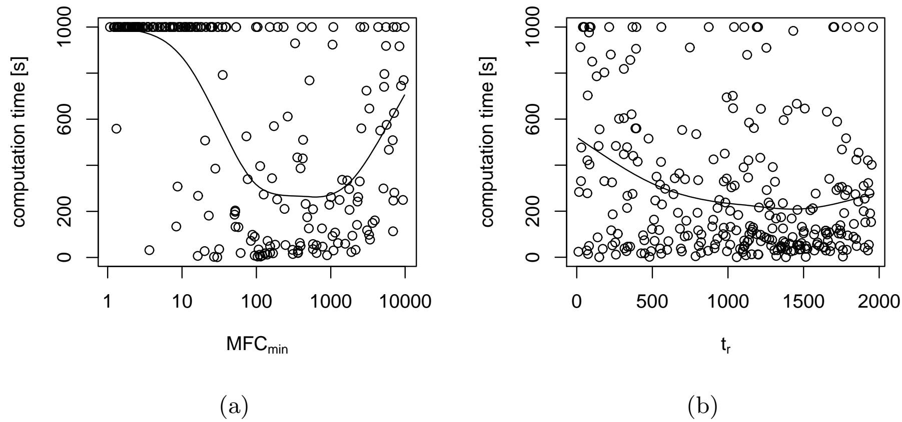
We see a hyperbolic tangent that serves as a threshold function. It is centred around $\frac{\mathrm{MSE}(\mathcal{T})}{\mathrm{MSE}(\mathcal{B})} = 1$. The threshold steepness is regulated by $t_s = 10$.

Let $\mathrm{MFC}_{\mathrm{min}}$ and t_r be adjusted in a test of a relatively complex function, in order to tune the evolutionary method towards more advanced tasks. We will use a set of samples $g^L(x)$, described in detail further in Sect. 6. The parameters in question will be tuned to minimise the average computation time t of the evolutionary method. To save on time, each test will be terminated whenever $t = 1000$. Figure 2(a) shows, that $\mathrm{MFC}_{\mathrm{min}} = 500$ is approximately optimal. Let us then, having that value, test a range of values t_r – a respective diagram in Fig. 2(b) shows a respective optimum value of about $t_r = 1500$.

A diagram of (5), using the adjusted parameters, is shown in Fig. 3.

6 Tests

Let us take advantage of the property of the presented method, that it is not limited to a polynomial approximation, and chose a function to generalise $g(x)$



(a) (b)

Fig. 2. Diagrams of computation time against (a) different MFC_{min}; for each sample, t_r is randomly picked using uni$(0, 2000)$; (b) different t_r and $\text{MFC}_{\text{min}} = 500$. The solid lines are smooth approximations.

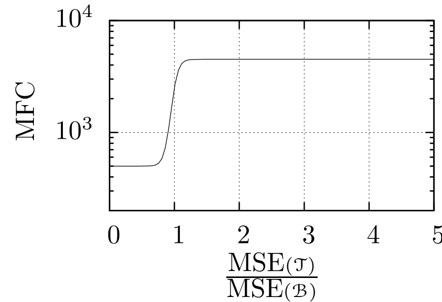


Fig. 3. A diagram of maximum failure count, given the ratio of MSE between the exploited candidate and the best one.

that contains a periodic function. Let us also make $g(x)$ relatively non–trivial for an evolutionary approach, by making the diagrams of its sub–components not resembling $g(x)$ – this may make it less possible, that such a sub–component alone would be considered interesting enough in the exploitation stage to be gradually completed by other sub–components. We will also model $g(x)$ so that along a certain part of its domain it is very similar to a function $y = x^2$. The parabola will serve as a honey–pot, that will attempt to distract our evolutionary method by a deep and, thanks to the symbolic simplicity of the honey pot, easily reachable local minimum. Let $g(x)$ be

$$y = 8\big(1 + \sin(0.59x + 4.6)\big). \tag{6}$$

The diagram in Fig. 4 confirms, that the sub–functions of $g(x)$ are dissimilar to $g(x)$, fitting–wise.

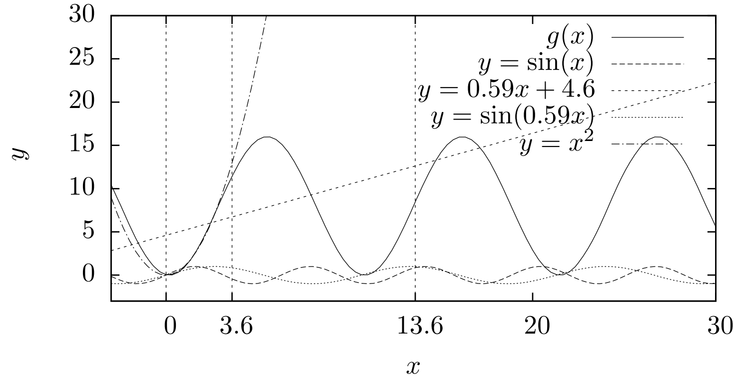


Fig. 4. A diagram of $f(x)$, some of its sub-components, and the honey–pot parabola.

An ideal generalising program is shown in Fig. 5. There are 9 instructions in

```
0   <1> PUSH 0.59
1   <1> MUL
2   <1> PUSH 4.6
3   <1> ADD
4   <1> SIN
5   <1> PUSH 1.0
6   <1> ADD
7   <1> PUSH 8.0
8   <1> MUL
```

Fig. 5. A program representing exactly $g(x)$.

the program, but we will allow a larger $P_n = 15$, as it is unlikely, that such a compact candidate will be found, especially without any preceding intermediate stages with a larger number of instructions. The program requires only two elements in the stack, but let the stack size be $P_s = 10$ because of the reasons similar to the above ones.

Let there be two sets of samples for the evolutionary method. A 'short' set $g^S(x)$ contains 30 samples of $g(x)$ only from within a domain fragment $S = \langle 0.1, 3.6 \rangle$, and a 'long' set $g^L(x)$ contains 30 samples from $L = \langle 0.1, 13.6 \rangle$. Both fragments start from 0.1 and not from 0, so that a lesser percentage of candidates become invalid because of a possible division by zero.

To convey the computational efficiency, any of the tests will be limited with a maximum computation time of 1000 seconds, and a number of tests that succeeded in fulfilling this criterion will be given.

6.1 Extrapolation of an inflection

We see that $g^S(x)$ is very similar to the honey–pot, and as the honey–pot lacks any inflection point, the hint to the evolutionary method that $g(x)$ has an inflection point within S would be rather subtle. Let us use the presented method to

extrapolate $g(x)$ outside S, to see if the method was sensitive to that subtle hint. Let $\mathrm{MSE}_{\max} = 1 \cdot 10^{-5}$ be rather small, as $g^S(x)$ is easy to fit by the discussed method.

Out of the 20 tests run, 6 met the 1000 seconds criterion. Their generalising diagrams are illustrated in Fig. 6. Five of these could be described as reasonable

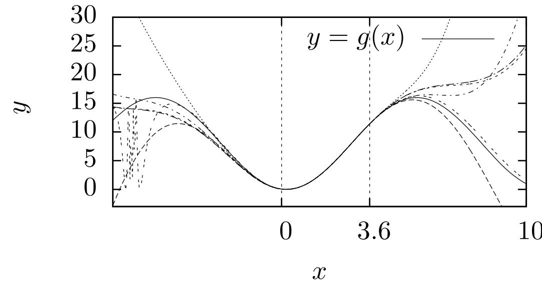


Fig. 6. A diagram $g(x)$, and of a set of 6 functions fitting to $g^S(x)$ at $\mathrm{MSE}_{\max} = 1{\cdot}10^{-5}$.

extrapolations, given the limited domain fragment covered by $g^S(x)$ – various interpretations of the slopes, of a stationary point and of a deflection point are seen. We see that the evolutionary method was able to propose interpretations very different from the honey–pot function. The sixth extrapolations contains garbage to the left of $g^S(x)$.

6.2 Extrapolation of periodicity

The set $g^L(x)$ presents a strong hint of being periodical. We expect the genetic programs to finely extrapolate that periodicity. Let $\mathrm{MSE}_{\max} = 1 \cdot 10^{-4}$ be larger than the error threshold used in the previous test, as $g^L(x)$ turns out to be relatively more difficult to fit. This time, out of 20 tests run, all met the 1000 seconds criterion. The generalising diagrams of the first 10 tests are illustrated in Fig. 7. We see a visually almost perfect fit, despite the relaxed value of $\mathrm{MSE}_{\max}$. A fitting error shows, though, that the generalising programs are not identical to $g(x)$, and that the fitting quality decreases on average as the extrapolation distance increases. Browsing the code of some of the respective final programs, depicted in Fig. 8, confirms, that $g(x)$ appears to be simpler than the generalising programs.

7 Discussion

If we look at the code of the final candidates, shown in Fig. 8, it is seen, that they still contain instructions of very different strengths, even if the generalised function could be defined only in terms of instructions having a binary strength of 0 or 1. The fractional property of the programs serves thus not only the purpose

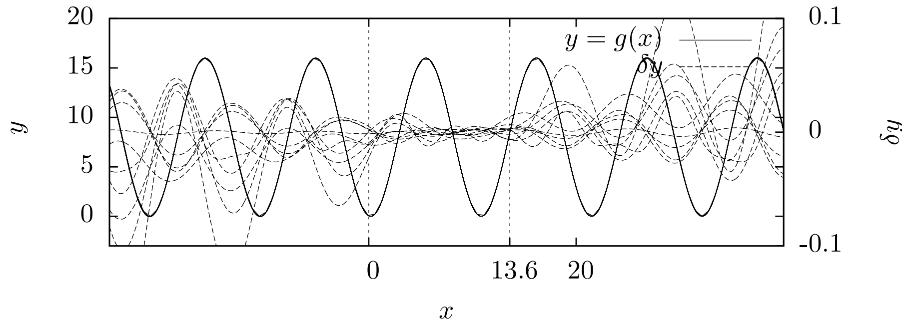


Fig. 7. A diagram $g(x)$, a set of 10 functions fitting to $g^L(x)$ at $\mathrm{MSE}_{\max} = 1 \cdot 10^{-4}$, and a fitting error δy of each.

of a continuous evolution, but also extends the expressivity of the candidates. Consider e.g. the following program: ⟨1⟩ PUSH x; ⟨0.5⟩ MUL, which computes x^2, even that x is pushed only once, and no power instruction is present.

As the programs utilise common algebraic operators and common mathematical functions, a question might be raised, if elegant symbolic equations could be extracted from them. If a generalising program appears to be closely fitting, yet representing a much complex equation that the generalised function, like in the case of the test in Sect. 6.2, then perhaps a decrease of $\mathrm{MSE}_{\max}$ might refine the programs up to the point of making some of the fractional instructions effectively almost disappear, by making their strengths very close to zero. An algorithm simplifying a symbolic equation might then be applied, which would hypothesise, that such instructions are redundant, and would remove them completely before simplifying the rest of the program in order to extract a symbolic equation.

References

1. Alba, E., Dorronsoro, B.: The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. Evolutionary Computation, IEEE Transactions on 9(2), 126–142 (2005)
2. Charbonneau, P.: Release notes for pikaia 1.2. Tech. rep., NCR/NT-451+ STR, NCR Technical Note, Boulder, Colorado (2002)
3. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: 1st International Conference on Genetic Algorithms. pp. 183–187. Carnegie-Mellon University, Pittsburgh, PA, USA (1986)
4. Eiben, A., Schippers, C.: On evolutionary exploration and exploitation. Fundamenta Informaticae 35(1-4) (1998)
5. Freitas, R., Gómez-Marín, C., Wilson, J.M., Casares, F., Gómez-Skarmeta, J.L.: Hoxd13 Contribution to the Evolution of Vertebrate Appendages. Developmental Cell 23(6), 1219–1229 (2012)
6. Goldberg, D.E.: Real-coded genetic algorithms, virtual alphabets, and blocking. Complex Systems 5, 139–167 (1990)

```
0    <0.5775> MUL
1    <0.3946> PUSH -6.7771
2    <0.9771> POP
3    <0.1400> PUSH 6.8841
4    <0.1872> NEG
5    <0.5722> SIN
6    <0.2324> PUSH 7.4358
7    <0.5234> COPY 2:1.6608
8    <0.1365> PUSH 8.9045
9    <0.1909> PUSH 6.7019
10   <0.2116> PUSH -5.8994
11   <0.2126> NEG
12   <0.8238> MUL
13   <0.2401> ADD
14   <0.1423> ADD
```

```
0    <0.3185> PUSH 7.1064
1    <0.2389> NEG
2    <0.0788> PUSH -6.3361
3    <0.9691> SIN
4    <0.4833> ADD
5    <0.2392> ADD
6    <0.8587> COPY 1:1.1385
7    <0.4592> PUSH 7.4412
8    <0.3479> MUL
9    <0.8930> PUSH 1.8314
10   <0.4190> POP
11   <0.8485> COPY 1:1.1542
12   <0.0793> COPY 1:1.1391
13   <0.0841> ADD
14   <0.0164> COPY 1:0.6350
```

```
0    <0.6323> ADD
1    <0.1373> NEG
2    <0.2402> PUSH -9.4909
3    <0.6012> NEG
4    <0.8460> COPY 1:0.1593
5    <0.1655> PUSH -4.3535
6    <0.2287> COPY 2:1.2507
7    <0.2647> PUSH 6.4488
8    <0.8651> SIN
9    <0.1737> POP
10   <0.3067> PUSH -4.2082
11   <0.6459> PUSH -7.4563
12   <0.8002> MUL
13   <0.1821> ADD
14   <0.1140> ADD
```

```
0    <0.9452> PUSH -9.3873
1    <0.6560> PUSH -5.0818
2    <0.2104> PUSH -5.6119
3    <0.4147> INV
4    <0.7736> COPY 1:1.3795
5    <0.6274> POP
6    <0.3134> COPY 2:0.5906
7    <0.6651> COPY 1:0.9763
8    <0.6518> SIN
9    <0.4214> POP
10   <0.8910> SIN
11   <0.4455> ADD
12   <0.2546> PUSH 7.0741
13   <0.4902> COPY 1:1.7485
14   <0.3605> MUL
```

Fig. 8. Example programs fitting to $g^L(x)$. To save space, digits only up to the 4th decimal place are shown.

7. Herrera, F., Lozano, M., Sánchez, A.M.: A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study. International Journal of Intelligent Systems 18(3), 309–338 (2003)
8. Kenneth, H., Robbins, K.A., von Ronne, J.: FIFTH: A stack based GP language for vector processing. In: EuroGP'07 Proceedings of the 10th European Conference on Genetic Programming. vol. 1, pp. 102–113. Springer–Verlag Berlin, Heidelberg, Valencia, Spain (2007)
9. Lin, L., Gen, M.: Auto-tuning strategy for evolutionary algorithms: balancing between exploration and exploitation. Soft Computing 13(2), 157–168 (2009)
10. Lindholm, T., Yellin, F.: Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc. (1999)
11. Oltean, M., Groşan, C., Dioşan, L., Mihăilă, C.: Genetic programming with linear representation: a survey. International Journal on Artificial Intelligence Tools 18(02), 197–238 (2009)
12. Perkis, T.: Stack-based genetic programming. In: Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on. vol. 1, pp. 148–153. Orlando, FL, USA (1994)
13. Smart, W., Zhang, M.: Continuously evolving programs in genetic programming using gradient descent. In: Mckay, R.I., Cho, S.B. (eds.) Proceedings of The Second Asian-Pacific Workshop on Genetic Programming. Cairns, Australia (2004)
14. Wright, A.H., et al.: Genetic algorithms for real parameter optimization. In: Foundations of Genetic Algorithms. pp. 205–218 (1990)