Chapter 1

EVOLUTION OF AN EFFECTIVE BRAIN-COMPUTER INTERFACE MOUSE VIA GENETIC PROGRAMMING WITH ADAPTIVE TARPEIAN BLOAT CONTROL

Riccardo Poli[1], Mathew Salvaris[1], and Caterina Cinel[1]
[1]*School of Computer Science and Electronic Engineering, University of Essex, Wivenhoe Park, CO4 3SQ, UK*

Abstract The Tarpeian method for bloat control has been shown to be a robust technique to control bloat. The covariant Tarpeian method introduced last year, solves the problem of optimally setting the parameters of the method so as to achieve full control over the dynamics of mean program size. However, the theory supporting such a technique is applicable only in the case of fitness proportional selection and for a generational system with crossover only. In this paper, we propose an adaptive variant of the Tarpeian method, which does not suffer from this limitation. The method automatically adjusts the rate of application of Tarpeian bloat control so as to achieve a desired mean program size. We test the method in a variety of standard benchmark problems as well as in a real-world application in the field of Brain Computer Interfaces, obtaining excellent results.

Keywords: Brain Computer Interfaces, Adaptive Tarpeian method, Bloat control

1. Introduction

Many techniques to control bloat have been proposed in the last two decades (for recent reviews see (Poli et al., 2008; Luke and Panait, 2006; Silva and Costa, 2009; Alfaro-Cid et al., 2010)). One of these is the Tarpeian method introduced in (Poli, 2003). The method is extremely simple in its implementation. All that is needed is a wrapper for the fitness function like the following:

```
if size(program) > average_program_size and random() < p_t then
  return( f_bad );
else
  return( fitness(program) );
```

were p_t is a real number between 0 and 1, `random()` is a function which returns uniformly distributed random numbers in the range $[0, 1)$ and f_{bad} is a constant which represents an extremely low (or high, if minimising) fitness value such that individuals with such fitness are almost guaranteed not to be selected.

The Tarpeian method has been used in a variety of studies and applications. For example, in (Mahler et al., 2005) its performance and generalisation capabilities were studied, while it was compared with other bloat-control techniques in (Luke and Panait, 2006; Wyns and Boullart, 2009; Alfaro-Cid et al., 2010). The method has been used with success in the evolution of bin packing heuristics (Burke et al., 2007; Allen et al., 2009), in the evolution of image analysis operators (Roberts and Claridge, 2004), in artificial financial markets based on GP (Martinez-Jaramillo and Tsang, 2009), just to mention a few.

In all cases the Tarpeian method has been a solid and efficient choice. A particularly nice feature of this method is that, because the wrapper *does not* evaluate the individuals being given a bad fitness, the higher the anti-bloat intensity, p_t, the faster the algorithm. All studies and applications, however, have had to determine by trial and error the value of the parameter p_t such that the size of programs would not exceed a given range of sizes. In other words, for a new application there was no way of saying *a priori* what values of p_t corresponded to what values of mean program size.

Recent research (Poli, 2010) has developed a technique, called *covariant Tarpeian method*, that allows one to dynamically and optimally set the rate of application of the Tarpeian method, p_t, in such a way as to completely control the evolution of the mean program size. The method has excellent properties but also some important drawbacks which we seek to overcome in this paper.

We, therefore, start the paper with a brief review of the covariant Tarpeian method (Section 2). This is followed by our proposed adaptive technique in Section 3. We test the algorithm on a variety of benchmark problems in Section 4. We propose our real-world application in the domain of Brain Computer Interfaces (BCI) in Section 5. We provide our conclusions in Section 6.

2. Covariant Tarpeian Method

The covariant Tarpeian method was inspired by the size evolution equation developed in (Poli, 2003; Poli and McPhee, 2003). As shown in (Poli and McPhee, 2008), this can be rewritten as follows

$$E[\Delta\mu] \quad = \quad \frac{\mathrm{Cov}(\ell, f)}{\bar{f}} \tag{1.1}$$

where E is the expectation operator, $\Delta\mu$ is the change in average program size from this generation to the next, ℓ is program size, f is fitness and $\bar{f}$ is the average fitness in the current generation. This equation applies if fitness proportionate selection is used in a generational system with independent selection and symmetric sub-tree crossover.

To model the effects on program size of the Tarpeian method in a GP system (Poli, 2010) specialised this equation and derived the following approximation:

$$E[\Delta\mu_t] \cong \frac{\mathrm{Cov}(\ell, f) - p_t\phi_> \left[\mathrm{Cov}_>(\ell, f) + (\mu_> - \mu)(\bar{f}_> - f_{bad})\right]}{\bar{f} - p_t\phi_>(\bar{f}_> - f_{bad})} \tag{1.2}$$

where $\Delta\mu_t$ is the change in average program size from one generation to the next *in the presence of Tarpeian bloat control*, $\phi_>$ is the proportion of above-average-size programs, $\bar{f}_>$ is the average fitness of such programs, $\mu_>$ is the average size of such programs, and $\mathrm{Cov}_>(\ell, f)$ is the covariance between program size and fitness within such programs.[1]

With this explicit formulation of the expected size changes, we can find out for what value of p_t we get $E[\Delta\mu_t] = 0$. By setting the l.h.s. of Equation (1.2) to 0 and solving for p_t, we obtain:

$$p_t \cong \frac{\mathrm{Cov}(\ell, f)}{\phi_> \left[\mathrm{Cov}_>(\ell, f) + (\mu_> - \mu)(\bar{f}_> - f_{bad})\right]}. \tag{1.3}$$

This equation allows one to determine how often the Tarpeian method should be applied to modify the fitness of above-average-size programs as a function of a small set of descriptors of the current state of the population and of the parameter f_{bad}.

An important limitation of these results is that they can be applied only to *generational* systems with *fitness proportionate selection*. Also, they are applicable only to systems with symmetric crossover and *no mutation*.

[1] Note that because size distributions are typically skewed, the proportion of above-average-size programs, $\phi_>$, is not necessarily 0.5. For example, in a population containing 5 programs of size 1, 4 of size 2, 3 of size 3, 2 of size 4 and 1 of size 5, $\phi_> = \frac{2}{5}$.

While there are many GP systems that fit this particular specification, many more, perhaps most, do not. In particular, the preferred selection strategy in GP is tournament selection (particularly when large populations are used, since it requires a computation load which is linear in the population size, while fitness proportionate selection is quadratic). Also, steady state systems are very frequently used in GP, since they avoid the overhead of storing two generations.

For all these systems, we need a different recipe to adapt p_t.

3. Adaptive Tarpeian Algorithm

The recipe we propose to set p_t to achieve this is very simple. We initially set $p_t = 0$. Then, at each generation, if the mean (or median) size of the individuals in the population is bigger than a user-settable threshold T, we increment p_t by an amount α, otherwise we decrement p_t by an amount α, where α is a constant < 1. In these operations, p_t is clipped within the interval $[0, 1]$. As we will show later, the threshold T can be modified from generation to generation to follow a particular schedule — for example, $T(t) = 50 + 25 \times \sin(t/10)$ — but will often be a constant.

Let us try to understand the rationale for this algorithm. We will do this by considering a generational system with fitness proportionate selection and we will apply the method to mean program sizes so as to be able to refer to the equations presented in the previous section for explanations. However, similar processes happen also in steady state algorithms, in systems which use forms of selection other than fitness proportionate and for median program sizes.

Since initially $p_t = 0$, if the user sets T to be bigger than the initial mean program size, for as long as the mean program size is below the threshold T, p_t remains zero. So, there is no Tarpeian bloat control and size can evolve freely. In a generational system using fitness proportionate selection mean program size will, therefore, obey Equation (1.2) with $p_t = 0$, i.e., Equation (1.1).

In the presence of bloat, at some point $\mathrm{Cov}(\ell, f)$ will be sufficiently high for sufficiently long to cause the mean program size to surpass the threshold T. At this point the algorithm start increasing p_t thereby initially partially and then completely compensating this positive covariance (see numerator of Equation (1.2)). Normally it will take some generations to reach complete compensation, i.e., for $E[\Delta\mu_t]$ to become zero. By this time the mean program size will have exceeded T and therefore the algorithm continues to increase p_t, thereby overcompensating for the covariance and causing the mean program size to start

dropping. Since p_t continues to be incremented until the mean program size is below T, eventually the mean program size will become less than T and in fact will continue to decrease for a while, until p_t is sufficiently small not to compensate for the covariance any more. At this point, in the presence of a positive covariance, the cycle will repeat (typically dampened). Naturally, the frequency of these oscillations will depend on the choice of α with the bigger its value the quicker the cycles.

Other than for the aforementioned effect, as we will see, the behaviour of the algorithm is little influenced by α. However, we still have one degree of freedom to fix: the parameter T. This is not different from other (non-covariant) bloat control methods. In the parsimony pressure method, for example, one needs to fix the parsimony coefficient, while in the original Tarpeian method one had to fix the rate of application of the operator. The difference between such methods and the adaptive Tarpeian method is that with the latter it is possible to precisely prescribe what size programs should have on average at each generations, while for the former the mapping from parameters to sizes is application-dependent and generally unknown.

4. Results on Benchmark Problems

GP Setup and Benchmark Problems

In order to extensively test the adaptive Tarpeian method we considered four symbolic regression problems. In three of the problems — the quartic, quintic and sextic polynomial problems — the objective was to evolve a function which fits a polynomial of the form $x + x^2 + \cdots + x^d$, where $d = 4, 5, 6$ is the degree of the polynomial, for x in the range $[-1, 1]$. In the fourth problem — the unequal-coefficients quartic polynomial problem — the objective was to fit the polynomial $x + 2x^2 + 3x^3 + 4x^4$. Fitness cases for all problems were obtained by sampling the corresponding polynomials at the 21 equally spaced points $x \in \{-1, -0.9, \ldots, 0.9, 1.0\}$.

We used a tree-based GP system implemented in Python with all numerical calculations done using the Numpy library (which is implemented in C). The system uses a steady-state update policy. It evolved a population of either 100 or 1,000 individuals with tournament selection with a tournament size of 5, the grow method with a maximum initial depth of 4, sub-tree crossover applied with a rate of 0.9 and sub-tree mutation with a rate of 0.1. Both used a uniform selection of crossover/mutation points. Runs lasted 50 generations unless otherwise stated. The system used the primitive set shown in Table 1-1.

Table 1-1. Primitive set used in our tests with polynomial regression problems.

Primitive	*Arity*	*Functionality*
x	0	Independent variable
$-1.0, -0.8, \ldots, 1.0$	0	Numerical constants
sin, cos	1	Sine and cosine functions
*, +, −	2	Standard arithmetic operators
%	2	Protected division. It returns 1 if the denominator is 0; $\min(\max(\frac{\text{num}}{\text{den}}, 10^{-5}), 10^5)$ otherwise.
if	3	If function. If the first argument is non-zero it returns its second argument, otherwise the third.

We tested the system with and without adaptive Tarpeian bloat control. When the method was used, we tested it for $\alpha \in \{0.05, 0.1, 0.25, 0.5, 1.0\}$ and for a threshold $T = 30$. In each configuration we performed 30 independent runs.

Polynomial Regression Results

In Figure 1-1 we show the evolution of the median (across 30 independent runs) of the best-of-generation fitness and of the median program size in runs with the Quartic polynomial problem for different values of the adaptive Tarpeian method parameter α and for populations of 1,000 individuals. In Figure 1-2 we show the corresponding results for the Sextic polynomial. For space limitations we do not report the results for the Quintic polynomial and the Quartic with unequal coefficients, but we note that results were qualitatively similar. Results with smaller populations of 100 individuals (not reported) were also similar in terms of program size dynamics, although best fitnesses were naturally significantly worse.

As we can see from Figures 1-1 and 1-2 there are no significant differences in the best-of-generation fitness dynamics between the case where no bloat control is exerted and in the case of adaptive Tarpeian bloat control. This is also confirmed by the success rate figures reported in Table 1-2.[2]

[2]While the results reported in Table 1-2 provide a useful indication of relative performance differences, they were obtained for one particular definition of success: that the fitness is no greater than 1. Naturally, different definitions of success may produce different results.

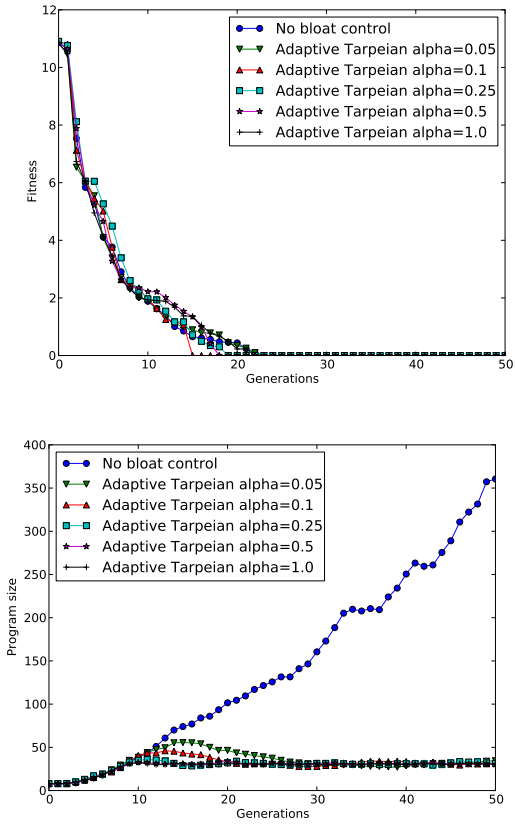


Figure 1-1. Evolution of the best-of-generation fitness and the median program size in the Quartic polynomial problem with different values of α and for a population of 1,000 individuals. Data points are medians across 30 independent runs.

However, looking at the median size dynamics in Figures 1-1 and 1-2 we see that at all values of α tested the method keeps the median size of the individuals in the population under tight control, while in the absence of bloat control we see significant bloat. As a result runs lasted between 10 (for the Quartic polynomial) and 20 (for the Quartic polynomial with unequal coefficients) times more without bloat control. What differs as α is varied is the duration of the transient which takes place when the threshold $T = 30$ is first hit, with the smallest values of α being affected by the largest overshoots in program size.

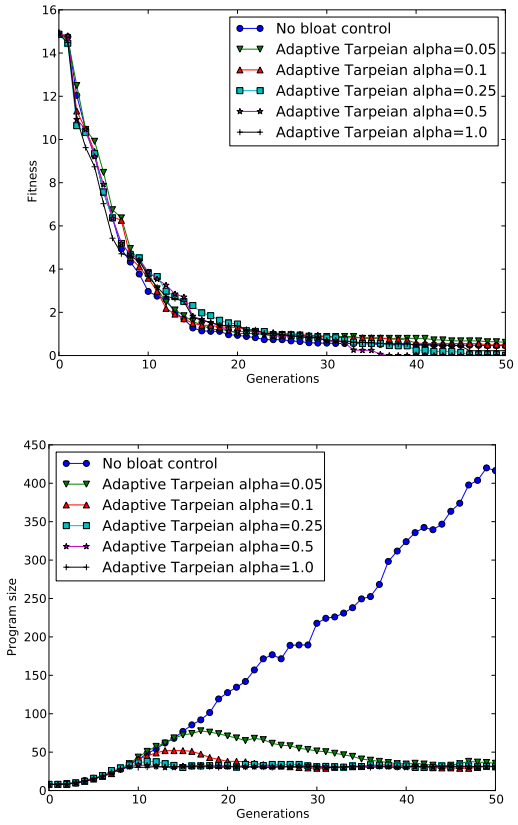


Figure 1-2. Evolution of the best-of-generation fitness and the median program size in the Sextic polynomial problem with different values of α and for a population of 1,000 individuals. Data points are medians across 30 independent runs.

5. Results on a BCI Mouse Problem

The problem

Brain-computer interfaces convert signals generated from the brain into control commands for devices such as computers. Such systems are often based on the analysis of brain electrical activity obtained via electroencephalography (EEG). Within EEG-based BCIs, the analysis of event related potentials (ERPs) has been particularly effective. ERPs are electrical components of brain activity produced in response to external stimuli. Some such ERPs can be used to determine user intentions in BCIs by relating their presence to specific external stimuli.

Table 1-2. Success rates for different population sizes, benchmark problems in the presence and in the absence of adaptive Tarpeian bloat control with different values of α. Success was defined as a fitness no greater than 1. Success rate figures are percentages.

Population size	Problem polynomial	No bloat control	Adaptive Tarpeian with α				
			0.05	0.1	0.25	0.5	1.0
100	Quartic	50.0	33.3	56.7	**60.0**	43.3	43.3
100	Quintic	36.7	26.7	23.3	33.3	**40.0**	23.3
100	Sextic	**46.7**	10.0	26.7	30.0	30.0	43.3
100	Quartic Uneq.	**20.0**	3.3	0.0	3.3	6.7	10.0
1000	Quartic	96.7	96.7	96.7	**100.0**	96.7	96.7
1000	Quintic	93.3	86.7	96.7	**100.0**	**100.0**	96.7
1000	Sextic	90.0	76.7	**100.0**	90.0	90.0	**100.0**
1000	Quartic Uneq.	**66.7**	50.0	43.3	60.0	50.0	60.0

Given the point-and-click nature of most modern user interfaces, an important application of BCI is controlling 2–D pointer movements. Some initial success in developing an ERP-based mouse has been reported in (Citi et al., 2008) and a variety of alternatives to this scheme were explored in (Salvaris et al., 2010). However, in both cases only semi-linear transformations were used to transform EEG signals into mouse movements. These have obvious limitations, particularly in relation to noise and artefact rejection. So, we wondered if GP, with its ability to explore the huge space of computer programs, could produce even more powerful transformations while also performing feature selection and artefact handling at the same time.

Our BCI mouse uses visual displays showing 8 circles arranged around a circle at the centre of the display as in Figure 1-3(far left). Each circle represents a direction of movement for the mouse cursor. Circles temporarily changed colour – from grey to either red or green – for a fraction of a second. We will call this a *flash*. The aim was to obtain mouse control by mentally focusing on the flashes of the stimulus representing the desired direction and mentally naming the colour of the flash. Flashes lasted for 100 ms and the inter-stimulus interval was 0 ms. Stimuli flashed in clockwise order. This meant that the interval between two target events for the protocol was 800 ms. We used a black background, grey neutral stimuli and either red or green flashing stimuli.

Data from 8 participants with an average age of 33 were used. They all had normal or corrected normal vision except for subject 5 who had strabismus. Each session was divided into runs, which we will call *direction epochs*. Each participant carried out 16 direction epochs, this resulted in the 8 possible directions being carried out twice.

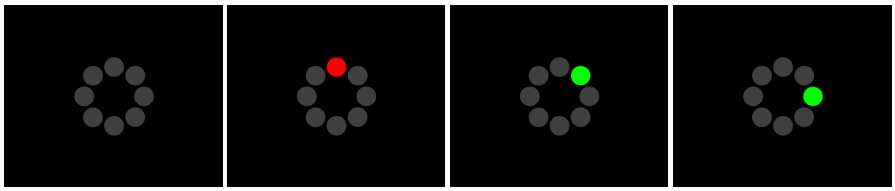


Figure 1-3. Stimuli used in our BCI experiments: initial display and three sequentially flashed stimuli.

Each direction epoch started with a blank screen and after 2 seconds the eight circles appeared near the centre of the screen. A red arrow then appeared for 1 second pointing to the target (representing the direction for that epoch). Subjects were instructed to mentally name the colour of the flashes for that target. After 2 seconds the flashing of the stimuli started. This stopped after 20 to 24 trials, with a trial consisting of the sequential activation of each of the 8 circles. In other words each direction epoch involves between $20 \times 8 = 160$ and $24 \times 8 = 192$ flashes. After the direction epoch had been completed, subjects were requested to verbally communicate the colour of the last target flash.

Data were sampled at 2048 Hz from 64 electrode sites using a BioSemi ActiveTwo EEG system. The EEG channels were referenced to the mean of the electrodes placed on either earlobe.

The data from each subject were used to train an ensemble of 2 hard-margin linear Support-Vector Machines (SVMs) for that subject. No channel selection was performed, and, so, all 64 channels were used. The data were filtered between 0.15 and 30 Hz and downsampled to 128 Hz. Then, from each channel an 800 ms epoch was extracted which was further decimated to 32 Hz.

The SVM classification results were estimated using 16 fold cross-validations, with each direction epoch being a cross-validation fold. After training, the output of each SVM can be interpreted as a degree of targetness of a stimulus as inferred from the feature vector associated with the stimulus.

The data produced by the SVM can be used to control the mouse. If θ is the angle corresponding to the last flashed stimulus (0° for the North direction, 45° for the North-East direction, etc.) and S is the corresponding score produced by the SVM, it is sufficient to apply a displacement $S \times (\sin\theta, \cos\theta)$ to the mouse cursor to obtain a reasonable trajectory. This is because the SVM tends to produce positive scores for epochs that represent targets and negative values for epochs that represent non-targets. However, the trajectories of the mouse thus produced are very convoluted and suffer enormously from the artifacts produced

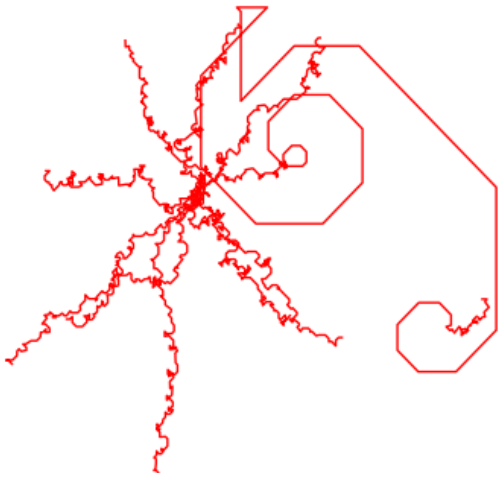


Figure 1-4. Example of normal and artifactual trajectories produced by the raw SVM output of one of our subjects. In all direction epochs shown the pointer starts in the middle of the screen and, ideally, is expected to move away from it at an angle of 0°, 45°, etc. and with a straight-line trajectory.

by swallowing, eye blinks and neck tension (which can easily generate voltages which are one or two orders of magnitude bigger than ordinary EEG signals). Examples of normal and artefactual trajectories produced by the raw SVM output are shown in Figure 1-4.

While the acquisition of data to train the SVMs of a subject and the training itself require only five minutes or so, making an SVM approach rather interesting, the particularly convoluted nature of the trajectories and the sensitivity of the SVMs to artifacts make the direct control of SVM raw output problematic. We, therefore, thought about using a program evolved by GP to post-process the output produced by the SVMs and to integrate this with the raw information available from the 64 EEG channels so as to straighten the trajectories produced by the raw SVM output and to make them insensitive to artifacts.

Naturally, GP cannot compete with the speed of SVMs and we cannot imagine subjects waiting for hours while GP finds a solution which suits their particular brain signals. So, the solution evolved by GP has to be general and reusable across subjects and across experiments.

GP Setup

For these experiments we used the same GP system as before, but in this case we switched on its strongly-typed features. Also, since fitness

evaluation in this domain of application is extremely computationally intensive, we also switched on its parallel feature which allows one to perform fitness evaluations across multiple CPU cores (via farming).

As before the system uses a steady-state update policy. It evolves a population of 1,000 individuals with tournament selection with a tournament size of 5, a strongly-typed version of the grow method with a maximum initial depth of 4, and strongly-typed versions of sub-tree crossover and sub-tree mutation. Both operators are applied with a 50% rate and use a uniform selection of crossover/mutation points. The system uses the primitive set shown in Table 1-3. Program trees were required to have a `Float` return type.

With this setup we performed runs of 50 generations. For Tarpeian we used a threshold $T = 20$ and $\alpha = 0.1$. We did several runs with this system. However, because of the extreme computational load required by our fitness evaluation and the complexity of the problem, here we only report the results of our best run. The run took approximately 6 *CPU days* to complete.

The fitness function we used measures the dissimilarity between the ideal trajectory and the actual trajectory produced by a program averaged over the direction epochs in our training set. The training set was formed by 80 direction epochs, obtained by taking the 39 most noisy direction epochs out of the 128 available from our subjects and mixing them with 41 less noisy direction epochs. Since each direction epoch consisted in an average of 176 trials and there were 80 such epochs, measuring the fitness of a program required executing the program for over 14,000 times. Being an error measure, fitness is, naturally, minimised in our system. We describe its elements below.

The actual trajectory produced by a program on a training epoch is obtained by iteratively evaluating the program, each time feeding the samples relating to a new flash into the `Fp1`, `AF7`, etc. terminals (which effectively act as a sliding window on the EEG) as well as the raw output produced on that epoch by the SVM ensemble. The output of the program, which, as noted above, is of type `Float`, is multiplied by a unit vector representing the direction corresponding to the stimulus that flashed on the screen. This produces a result of the form $(\Delta x, \Delta y)$ which is used as a displacement to be applied to the current mouse position.

As illustrated in Figure 1-5, the ideal trajectory for each direction epoch is obtained by sampling at regular intervals the line segment connecting the origin to a point along the desired direction. The point is chosen by computing the expected distance reached by the linear SVM trained on the data for subject. The ideal trajectory is sampled in such a way to have the same number of samples as the actual trajectory. The

Table 1-3. Primitive set used in our BCI mouse application.

Primitive	Output Type	Input Type(s)	Functionality
±`0.5`, ±`1`, ±`2`, ±`5`, ±`10`, `0`, ···, `25`	Float	None	Floating point constants used for numeric calculations and array indexing
`Fp1`, `AF7`, `AF3`, `F1`, ... (60 more channels)	Array	None	Returns an array of 26 samples following a flash from one of the channels. The samples are of type Float.
`SVM`	Float	None	Raw output of SVM ensemble.
`targetQ1`, `targetMed`, `targetQ3`, ... (6 more for non-targets and near-targets)	Float	None	First quartile, median and third quartile of the SVM scores for targets, near-targets and non-targets for a subject.
`+`, `-`, `*`, `min`, `max`	Float	(Float, Float)	Standard arithmetic operations plus maximum and minimum on floats.
>, <	Bool	(Float, Float)	Standard relational operations on floats
`if`	Float	(Bool, Float, Float)	If-then-else function. If the first argument evaluates to True, then the result of evaluating its second argument is returned. Otherwise the result of evaluating the third argument is returned.
`abs`	Float	Float	Returns the absolute value of a Float.
`mean`, `median`, `std`, `Amin`, `Amax`	Float	(Float, Float, Array)	Given a 26-sample array and two floats, treat the floats as indices for the array by casting them to integer and applying a modulus 26 operation. Then compute the mean, median, standard deviation, minimum or maximum, respectively, of the samples in the array falling between such indices (inclusive).

comparison between actual and ideal trajectory is then a matter of measuring the Euclidean distance between pairs of corresponding points in the two trajectories and taking an average. Notice that any detours from the ideal line and any slow-downs in the march along it in the actual trajectory are strongly penalised with our fitness measure.

In order to test the generality of evolved solutions we used a further 80 direction epochs obtained from a separate set of 5 subjects.

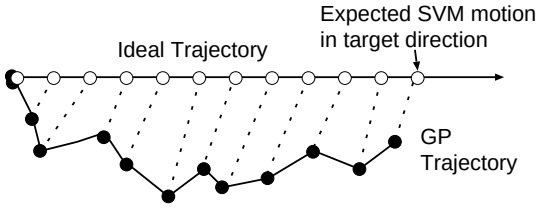


Figure 1-5. Ideal and actual trajectories used in the fitness calculation. Dashed lines indicate pairs of matching points. Fitness is the average distance between such points across trajectories.

BCI Mouse Results

Figure 1-6 shows the dynamics of median and best program's fitness as well as of median and best program's size in our runs. As one can immediately see, essentially there is no bloat, while fitness continues to improve, albeit quite slowly, for the 50 generations of the run.

The best evolved program is presented in tree form in Figure 1-7. To evaluate its performance we will compare its output to the output produced by the raw SVM data on our independent test set of 5 subjects.

Let us start from a qualitative analysis of the behaviour of this program. Figure 1-8(top) shows the output produced by the SVM ensemble for each of the direction epochs in the test set of 5 subjects (after the transformation of the scores into $(\Delta x, \Delta y)$ displacements). Note how convoluted trajectories are. Clearly while the SVMs get a good portion of the job done, namely turning the EEG signals into trajectories that are in *approximately* the right direction, they leave much to desire in terms of usability. Figure 1-8(bottom) shows the corresponding trajectories produced by our best evolved programs. Qualitatively it is clear that these trajectories are far less convoluted. Also, their end points appear to cluster more precisely near the desired directions of motion.

To quantitatively verify these observations, Table 1-4 shows a statistical comparison between the trajectories produced by GP and those produced by SVM in terms of mean, median, standard deviation and standard error of the mean of the distances between the ideal trajectory and the actual trajectory recorded in each of the 80 direction trials in our test set of 5 subjects. As we can see the evolved program produces trajectories which are on average closer to the ideal line than the corresponding trajectories produced by the SVMs. The differences are highly statistically significant as shown by a one-sided Wilcoxon signed rank test for paired data (p-value = 0.0009).

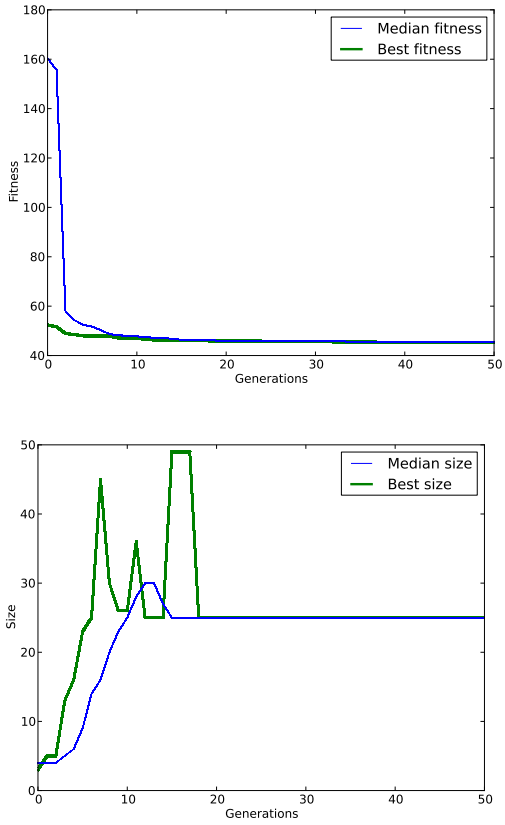


Figure 1-6. Plots of the median and best fitness and median and best program size vs generation number in our runs.

Table 1-4. Statistical comparison between SVM and the evolved solution for the BCI mouse problem.

Program	Mean	Median	Standard Deviation	Standard Error
GP	44.3389	41.5974	18.1760	2.0450
SVM	51.5003	50.9928	21.6596	2.4369

6. Conclusions

In this paper we have presented a variation of the Tarpeian method's for bloat control that allows the adaptation of the method main parameter — its rate of application p_t — in such a way to guarantee that programs don't grow bigger than a given threshold.

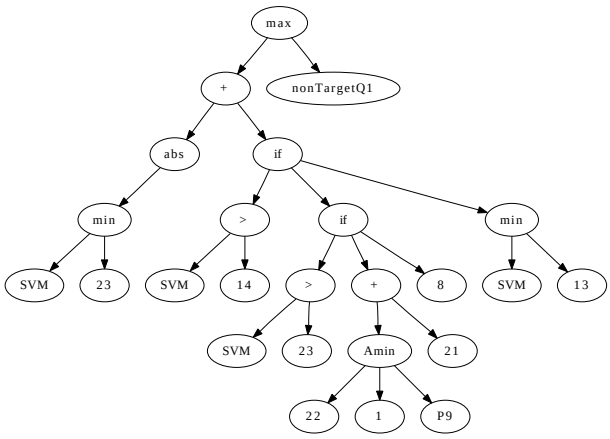


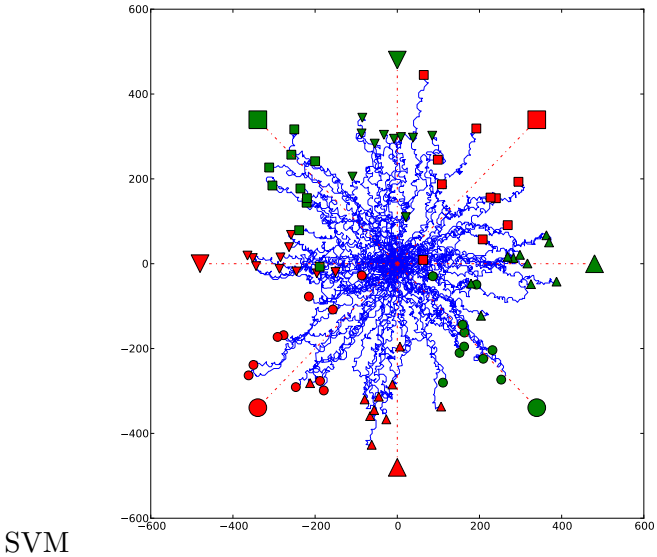
Figure 1-7. Best program evolved in our runs.

An earlier method — the covariant Tarpeian method introduced last year (Poli, 2010) — can solve this problem analytically. However, the method is only applicable to generational systems using fitness proportionate selection and relying only on crossover as their genetic operator. While the adaptive Tarpeian method is algorithmic and control of program size is asymptotic, it does not suffer from such limitations as we have demonstrated testing it with a steady state GP system with tournament selection and using a mix of sub-tree crossover (which is size-unbiased) and sub-tree mutation (which is size-biased).

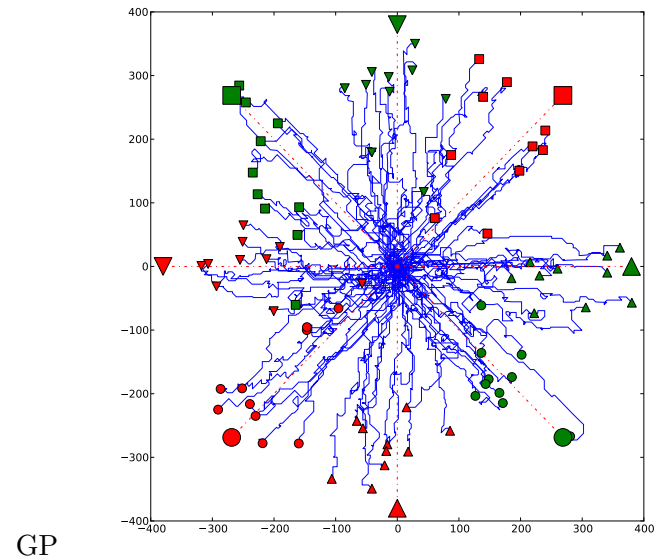
We tested the method on a variety of standard benchmark problems as well as in a real-world application — the evolution of a Brain Computer Interfaces mouse controller. Results with benchmark problems indicate that the method is very effective at keeping the size of programs under control without incurring any particular penalty in terms of end-of-run fitness achieved. Runs, however, lasted only a fraction of the time required without bloat control. The results in the BCI mouse problem were very good, the system being able to evolve a controller which is significantly better than a solution based on support vector machines. The solution also generalised very well to unseen data.

Acknowledgements

We would like to thank Luca Citi and the Engineering and Physical Sciences Research Council (grant EP/F033818/1).



SVM



GP

Figure 1-8. Graphical representation of the 80 sequences of SVM scores and the evolved-program scores for our 5 test subjects. The larger symbols connected via dotted lines to the origin represent the 8 desired directions of motion. The smaller symbols demarcate the end points of the corresponding pointer trajectories.

References

Alfaro-Cid, Eva, Merelo, J. J., Fernandez de Vega, Francisco, Esparcia-Alcazar, Anna I., , and Sharman, Ken (2010). Bloat control operators

and diversity in genetic programming: A comparative study. *Evolutionary Computation*, 18(2):305–332.

Allen, Sam, Burke, Edmund K., Hyde, Matthew R., and Kendall, Graham (2009). Evolving reusable 3D packing heuristics with genetic programming. In Raidl *et al.*, Guenther, editor, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 931–938, Montreal. ACM.

Burke, Edmund K., Hyde, Matthew R., Kendall, Graham, and Woodward, John (2007). Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In Thierens *et al.*, Dirk, editor, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1559–1565, London. ACM Press.

Citi, L., Poli, R., Cinel, C., and Sepulveda, F. (2008). P300-based BCI mouse with genetically-optimized analogue control. *IEEE transactions on neural systems and rehabilitation engineering*, 16(1):51–61.

Luke, Sean and Panait, Liviu (2006). A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344.

Mahler, Sébastien, Robilliard, Denis, and Fonlupt, Cyril (2005). Tarpeian bloat control and generalization accuracy. In Keijzer *et al.*, Maarten, editor, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 203–214, Lausanne, Switzerland. Springer.

Martinez-Jaramillo, Serafin and Tsang, Edward P. K. (2009). An heterogeneous, endogenous and coevolutionary GP-based financial market. *IEEE Transactions on Evolutionary Computation*, 13(1):33–55.

Poli, Riccardo (2003). A simple but theoretically-motivated method to control bloat in genetic programming. In Ryan, Conor, Soule, Terence, Keijzer, Maarten, Tsang, Edward, Poli, Riccardo, and Costa, Ernesto, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 204–217, Essex. Springer-Verlag.

Poli, Riccardo (2010). Covariant tarpeian method for bloat control in genetic programming. In Riolo, Rick, McConaghy, Trent, and Vladislavleva, Ekaterina, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, chapter 5, pages 71–90. Springer, Ann Arbor, USA.

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`.

Poli, Riccardo and McPhee, Nicholas (2008). Parsimony pressure made easy. In Keijzer *et al.*, Maarten, editor, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1267–1274, Atlanta, GA, USA. ACM.

Poli, Riccardo and McPhee, Nicholas Freitag (2003). General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206.

Roberts, Mark E. and Claridge, Ela (2004). Cooperative coevolution of image feature construction and object detection. In Yao *et al.*, Xin, editor, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 902–911, Birmingham, UK. Springer-Verlag.

Salvaris, Mathew, Cinel, Caterina, Poli, Riccardo, Citi, Luca, and Sepulveda, Francisco (2010). Exploring multiple protocols for a brain-computer interface mouse. In *Proceedings of 32nd IEEE EMBS Conference*, pages 4189–4192, Buenos Aires.

Silva, Sara and Costa, Ernesto (2009). Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179.

Wyns, Bart and Boullart, Luc (2009). Efficient tree traversal to reduce code growth in tree-based genetic programming. *Journal of Heuristics*, 15(1):77–104.