Natural Computing (2005) 4: 417–437
DOI: 10.1007/s11047-005-3666-7

Investigations into graceful degradation of evolutionary developmental software

PETER BENTLEY
Department of Computer Science, University College London, Gower St., London, WC1E 6BT, UK (E-mail: P.Bentley@cs.ucl.ac.uk)

Abstract. Today's software is brittle. A tiny corruption in an executable will normally result in terminal failure of that program. But nature does not seem to suffer from the same problems. A multicellular organism, its genes evolved and developed, shows graceful degradation: should it be damaged, it is designed to continue to work. This paper describes an investigation into software with the same properties. Three programs, one human-designed, one evolved using genetic programming, and one evolved and developed using a fractal developmental system are compared. All three calculate the square root of a number. The programs are damaged by corrupting their compiled executable code, and the ability for each of them to survive such damage is assessed. Experiments demonstrate that only the evolutionary developmental code shows graceful degradation after damage.

Key words: development, fractal, genetic programming, graceful degradation, software reliability

1. Introduction

Human designs are carefully crafted, consciously created fusions of experience and skill. Our designs usually work reliably and well under the conditions they were designed for. Unfortunately, they rarely work well under unforeseen conditions. A robot will fail on the wrong kind of ground; the traction of a car will fail on the wrong type of road; the reception of a cellular phone will be lost when in the wrong kind of surroundings. A program will fail in the wrong kind of software environment. And sadly for computer-users worldwide, the mess of different software on an average computer causes such complex environments that programs fail with tired regularity.

Natural systems are also carefully crafted, but there is no conscious mind or skill needed to produce nature's designs. Generations of past experience drives evolution to create robust, damage-tolerant solutions. Organisms don't fail if they sustain minor damage. Even a

heart attack can be recoverable. The equivalent damage to a human-designed program would always produce terminal failure.

The work described here continues an ongoing investigation into the use of developmental systems with evolutionary computation. Here, fractals are employed as a computer representation of proteins. Earlier work has shown that fractal proteins are highly evolvable by a genetic algorithm (Bentley, 2004; 2003c), that specific patterns of activation in a fractal gene regulatory network (GRN) can be evolved (Bentley, 2004, 2003b), that they can perform computational tasks such as function regression and robot control (Bentley, 2003a), and that evolved fractal GRNs naturally show fault-tolerance (Bentley, 2003c). This work now focuses on the evolution of developmental programs that display graceful degradation when damaged.

2. Background

Questions of reliability and graceful degradation occur frequently in fields focusing on embedded systems. To date, most solutions seem to depend on architectures that partition software into separate components, organised in such a way that the failure of non-critical components will not induce the failure of the whole system (Shelton and Koopman, 2001). Such approaches provide the most effective methods for achieving reliability, but they typically suffer from traditional problems of anticipation and detection – the designer of such systems must anticipate all types of fault and use that knowledge to design a robust system, and the system must be able to detect when a fault has occurred so that redundant components may take over. This work focuses on automatic systems that require no anticipation by designers of possible faults, and no detection by the system of the fault.

In Evolutionary Computation, scientists have been focussing on the ability of evolution, and more commonly developmental methods, to enable self-repairing behaviour and graceful degradation of solutions. Andy Tyrrell and his group create fault-tolerant hardware inspired by ideas of embryology and immune systems (Jackson and Tyrrell, 2002). More recently, Julian Miller has described experiments evolving developmental programs to create "French Flag" patterns (Miller and Banzhaf, 2003). He shows that development is able to regenerate these patterns should some of their cells be removed. Current work by Mahdavi and Bentley (2003) demonstrates how adaptive evolutionary

control can enable a "Smart Snake" to redevelop new movement strategies even after the loss of a crucial muscle (Nitinol wire).

In his research on fault-tolerant systems, Thompson (1997) describes how "graceful degradation for free" can be achieved in theory and in practice for robot controllers, "from the nature of the evolutionary process." Thompson suggests that mutation-insensitive individuals will, in the long term, survive better, thus producing a pressure towards fault-tolerant solutions. More recently, the same results were demonstrated with fractal developmental processes (Bentley, 2003c), where there are no direct mappings: pleiotropy and polygeny are prevalent, and genes are reused over many developmental iterations. It was shown that through the Baldwin Effect, solutions "naturally" became more efficient and fault-tolerant (Bentley, 2003c). In more detail, the work demonstrated that if evolution was permitted to run for a further 1000 generations after a perfect solution had evolved, the fractal GRNs continued to evolve: the number of genes and proteins that made up the solution was reduced (so there is less to be damaged), and duplicate genes were added, which provide redundancy and protection against damage.

This paper extends this work, showing that damage directly to the executable code (and not just a gene in the system) can be survived by evolved developmental programs.

3. Fractal proteins

Development is the set of processes that lead from egg to embryo to adult. Instead of using a gene for a parameter value as we do in standard EC (i.e. a gene for long legs), natural development uses genes to define proteins. If expressed, every gene generates a specific protein. This protein might activate or suppress other genes, might be used for signalling amongst other cells, or might modify the function of the cell it lies within. The result is an emergent, asynchronous, parallel "computer program" made from dynamically forming gene regulatory networks (GRNs) that control all cell growth, position and behaviour in a developing creature (Wolpert et al., 2001).

In this work, a biologically plausible model of gene regulatory networks is constructed through the use of genes that are expressed into *fractal proteins* – subsets of the Mandelbrot set that can interact and react according to their own fractal chemistry. Further

motivations and discussions on fractal proteins are provided in (Bentley, 2004, 2003a, b, c).

Table 1 describes the object types in the representation; Figure 1 illustrates the representation. Figure 2 provides an overview of the algorithm used to develop a phenotype from a genotype. Note how most of the dynamics rely on the interaction of fractal proteins. Evolution is used to design genes that are expressed into fractal proteins with specific shapes, which result in developmental processes with specific dynamics. Figures 3, 4, 5, 6 and 7 show examples of fractal proteins and their interactions. The full development algorithm is given in the Appendix.

The main equations used in the system are as follows:

Protein decay:

$$diffusedconc = prevconcentration * (1 - 1/\text{PROTEINDEC} + 0.2) \quad (1)$$

(PROTEINDEC is a constant normally set to 5).

Table 1. Types of objects in the model

Fractal proteins	Defined as subsets of the Mandelbrot set
Environment	Contains one or more fractal proteins (expressed from the environment gene(s)), and one or more *cells*
Cell	Contains a *genome* and *cytoplasm*, and has some *behaviours*
Cytoplasm	Contains one or more fractal proteins
Genome	Comprising *structural genes* and *regulatory genes*. In this work, the structural genes are divided into different types: *cell receptor genes*, *environment genes* and *behavioural genes*
Regulatory gene	Comprising operator (promoter or cis-site) region and coding (or output) region
Cell receptor gene	A structural gene with a coding region which acts like a mask, permitting variable portions of the environmental proteins to enter the corresponding cell cytoplasm
Environment gene	A structural gene which determines which proteins (maternal factors) will be present in the environment of the cell(s)
Behavioural gene	A structural gene comprising operator and cellular behaviour region

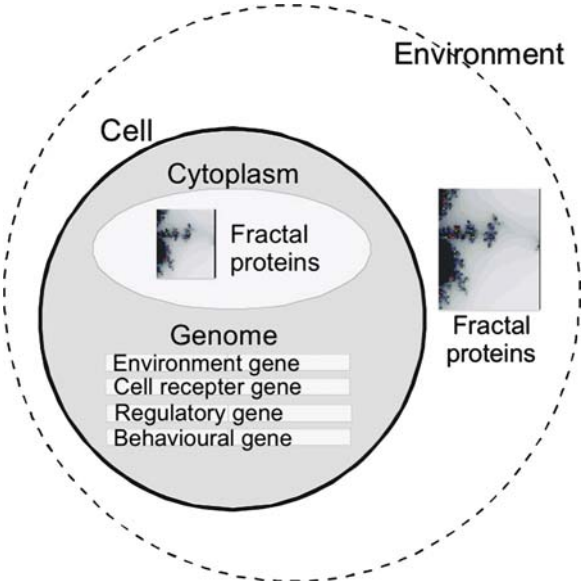

Figure 1. Representation using fractal proteins.

FRACTAL DEVELOPMENT

For every developmental time step:

For every cell in the embryo:

Express all environment genes and
calculate shape of merged environment fractal proteins

Express cell receptor genes as receptor fractal proteins
and use each one to mask the merged environment proteins
into the cell cytoplasm.

If the merged contents of the cytoplasm match a promoter
of a regulatory gene, express the coding region of the gene,
adding the resultant fractal protein to the cytoplasm.

If the merged contents of the cytoplasm match a promoter of a
behavioural gene, use coding region of the gene to specify a
cellular function.

Update the concentration levels of all proteins in the cytoplasm
If the concentration level of a protein falls to zero, that protein
does not exist.

Figure 2. The fractal development algorithm.

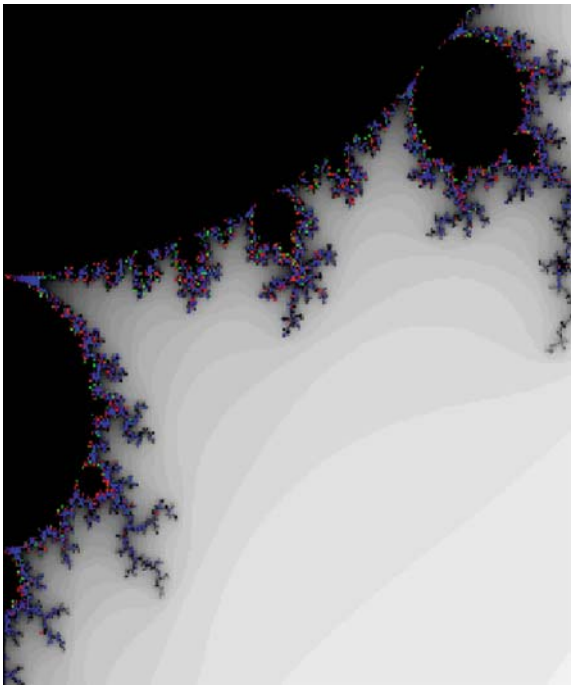

Figure 3. A fractal protein is a finite square subset of the Mandelbrot set (Mandelbrot, 1982), defined by three codons (x,y,z) that form the coding region of a gene in the genome of a cell. Each (x,y,z) triplet is expressed as a protein by calculating the square fractal subset with centre coordinates (x,y) and sides of length. Here we show an example of a fractal protein defined by ($x = 0.13254$, $y = 0.69812$, $z = 0.46830$).

Gene output concentration:

$$geneoutputconc = totalconc \times \tanh totalconc - (c_t/\mathrm{CWIDTH})/\mathrm{CINC} \quad (2)$$

where: *totalconc* is the mean concentration seen at the promoter, C_t is the concentration threshold from the gene promoter, CWIDTH is a constant (normally set to 30), and CINC is a constant (normally set to 2).

Gene activation probability:

$$activationprob = (1 + \tanh((matchnum - Affinitythreshold - C_t)/C_s))/2 \quad (3)$$

where: *matchnum* is the matching score, *Affinity threshold* is the matching threshold from gene promoter, C_t is a threshold constant (normally set to 50), and C_s is a sharpness constant (normally set to 50).

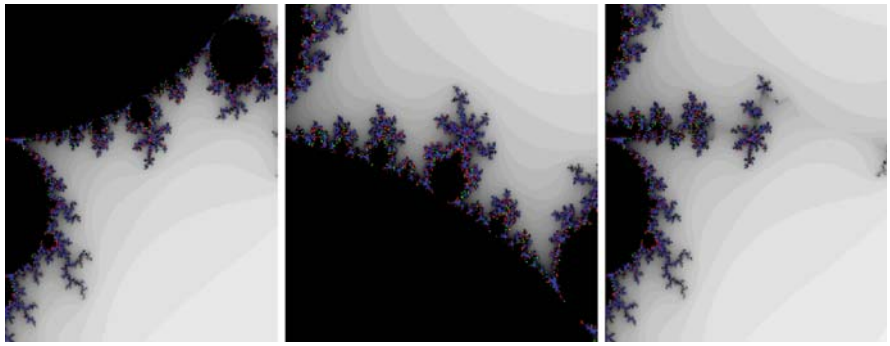

Figure 4. To model complex protein-protein and protein-gene interactions, fractal proteins interact according to their fractal shapes. The interaction occurs by merging separate protein shapes to form new, complex compounds. The result is a product of their own ''fractal chemistry'' which naturally emerges through the fractal interactions. Fractal proteins are merged (for each point sampled) by iterating through the fractal equation of all proteins in ''parallel'', and stopping as soon as the length of any is unbounded (i.e. greater than 2). Here we show two fractal proteins (left and middle) and the resulting merged fractal protein combination (right).

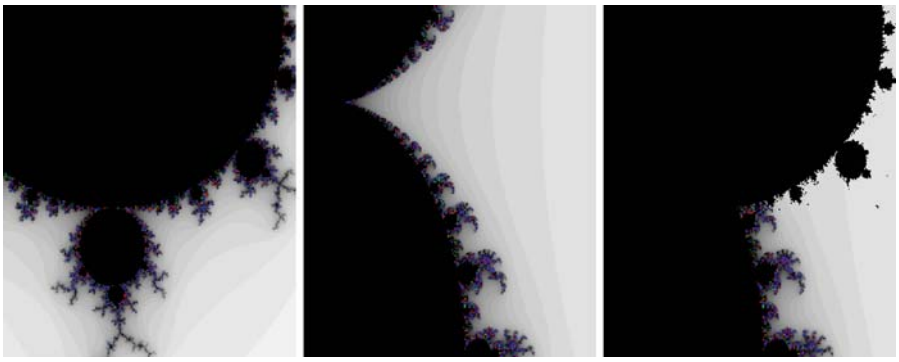

Figure 5. Fractal proteins can also act like a mask over other proteins, where all black regions of the mask are treated as opaque, and all other regions treated as transparent. Here we show a cell receptor protein (left), environment protein (middle), and the resulting masked protein to be combined with cytoplasm (right).

Behavioural gene output 1:
If the gene is being activated with a negative *Affinity threshold*,

$$output = output - (totalconcentration - concentrationthreshold) * fate \tag{4}$$

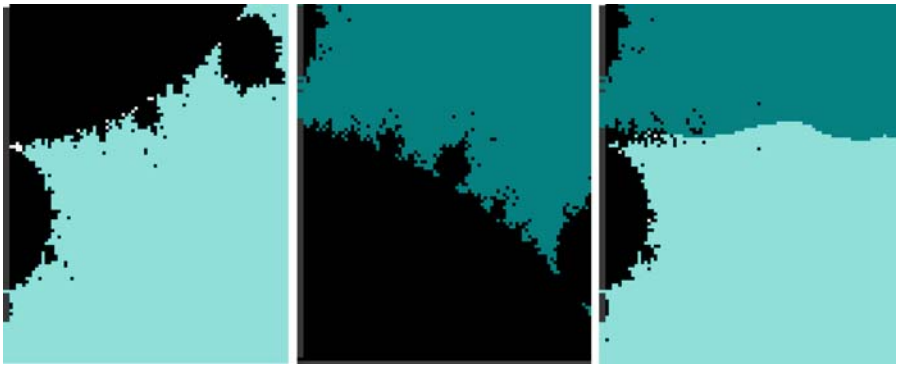

Figure 6. Each fractal protein also represents a certain *concentration* of protein (from 0 meaning "does not exist" to 200 meaning "saturated"). The total concentration of two or more merged fractal proteins is the mean of the different concentrations seen in their merged product. This Figure shows different concentrations of two fractal proteins (left and middle) and the concentration levels in their merged product (right).

Behavioural gene output 2:

If the gene is being activated with a positive *Affinity threshold*,

$$output = output + (totalconcentration - concentrationthreshold) * fate \tag{5}$$

3.1. *Genes*

The environment gene, cell receptor gene, regulatory genes, and behavioural genes all contain the following values: (*xp*,*yp*,*zp*, *Affinity threshold, Concentration threshold*) define the promoter (operator or precondition) for the gene and (*x*,*y*,*z*) define the coding region of the gene. A *type* value defines which type of gene is being represented, and can be one or all of the following: *environment*, *receptor*, *behavioural*, or *regulatory*. This enables the type of genes to be set independently of their position in the genome, enabling variable-length genomes. It also enables genes to be multi-functional, i.e. a gene might be expressed both as an environmental protein and a behaviour.

When *Affinity threshold* is a positive value, one or more proteins must match the promoter shape defined by (*xp*,*yp*,*zp*) with a difference equal to or lower than *Affinity threshold* for the gene to be activated. When *Affinity threshold* is a negative value, one or more proteins must match the promoter shape defined by (*xp*,*yp*,*zp*) with a difference equal

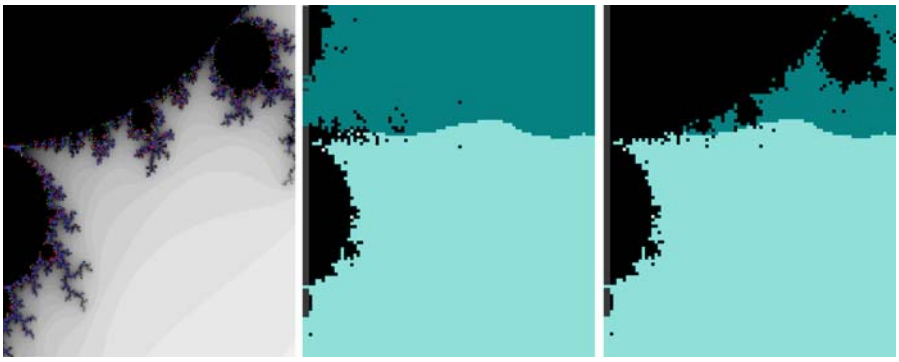

Figure 7. When being compared to the (*xp*,*yp*,*zp*) promoter region of a gene, the concentration seen on that promoter is described by all those regions that "fall under" the promoter. In other words, the merged product is masked by the promoter fractal, and the total concentration on the promoter is the mean of the resulting concentrations. We depict here the shape of the desired protein as defined by a promoter (left), the shape and concentration levels of merged proteins in the cytoplasm (middle) and the concentration levels seen on that promoter (right), where total concentration is taken as mean.

to or lower than |*Affinity threshold*| for the gene to be repressed (not activated).

To calculate whether a gene should be activated, all fractal proteins in the cell cytoplasm are merged (including the masked environmental proteins, see later) and the combined fractal mixture is compared to the promoter region of the gene.

The similarity between two fractal proteins (or a fractal protein and a merged fractal protein combination) is calculated by sampling a series of points in each and summing the difference between all the resulting values. (Black regions of fractals are ignored.) Given the similarity matching score between cell cytoplasm fractals and gene promoter, the activation probability of a gene is given by equation (3).

3.1.1. *Regulatory gene*

Should a regulatory gene be activated by other protein(s) in the cytoplasm (which have concentrations above 0) matching its promoter region, its corresponding coding region (*x*,*y*,*z*) is expressed (by calculating the subset of the Mandelbrot set) and new concentration level calculated. To do this, the concentration of the resulting protein is modified by incrementing with *geneoutputconc*, the result of a function of the concentration threshold (c_t) and the mean total concentration seen at the gene promoter (*totalconc*), as given in Section 3.5. In this

way, higher concentrations of protein on the promoter will cause an increased rate of output protein concentration growth, while lower concentrations (below the c_t threshold) will increase the diffusion rate of the output protein (its concentration will decrease at a higher rate).

The cell cytoplasm, which holds all current proteins, is updated at the end of the developmental cycle.

3.1.2. *Cell receptor gene*

At present, the promoter region of the cell receptor gene is ignored, and this gene is always activated. As usual, the corresponding coding region (x,y,z) is expressed by calculating the subset of the Mandelbrot set. However, the resultant fractal protein is treated as a mask for the environmental proteins.

3.1.3. *Environment gene*

Like the cell receptor gene, this gene is always activated. It produces environmental factors for all cells: fractal proteins of concentration 200. If there is more than one environmental gene, the expressed environmental proteins are merged before being masked by the receptor protein. If one or more values are being input to the system, the concentration of the environmental fractal proteins are set to those values, i.e. an input to the system disturbs the environment during development.

3.1.4. *Behavioural gene*

A behavioural gene is activated when other protein(s) in the cytoplasm match its promoter region (using the *affinity threshold*). For this application, a gradual activation between not activated and activated was required, using the x value of the coding region (x,y,z) triplet as a *fate* value to define a function, see equations (4) and (5).

Note how the total concentration of proteins seen on the promoter is offset against the *Concentration Threshold* gene and scaled by the *fate* gene (x value of the coding region), allowing evolution to adjust the range of values seen on the output, and used to specify behaviours. (If there is more than one behavioural gene, the change to *output* is averaged over all behavioural genes, each developmental step.)

3.2. *Development*

As was illustrated in Figure 2, an individual begins life as a single cell in a given environment. To develop the individual from this zygote

into the final phenotype, fractal proteins are iteratively calculated and matched against all genes of the genome. Should any genes be activated, the result of their activation (be it a new protein, receptor or cellular behaviour) is generated at the end of the current cycle.

All fractal calculations (masking, merging, comparisons) are performed by sampling the fractals at a resolution of 15×15 points. Note that the comparison is normally performed between the single fractal defined by (xp,yp,zp) of a gene and the merged combination of all other proteins currently in the cytoplasm. The fractal shape defined by the gene promoter is treated a little like the cell receptor mask – only those regions that are not black are actually compared with the contents of the cytoplasm.

Every developmental time step, the new concentration of each protein is calculated (synchronously). This is formed by summing two separate terms: the previous concentration level after diffusion (*diffusedconc*) and the new concentration output by a gene (*geneoutputconc*). These two terms model the reduction in concentration of proteins over time, and the production of new proteins over time, respectively; see equations (1) and (2).

Development continues for d cycles, where d is dependent on the problem. Note that if one of the cellular behaviours includes the creation of new cells, then development will iterate through all genes of the genome in all cells. Refer to the appendix for the full developmental algorithm.

3.3. *Evolution*

The genetic algorithm used in this work has been used extensively elsewhere for other applications (including GADES (Bentley, 1999)). A dual population structure is employed, where child solutions are maintained and evaluated, and then inserted into a larger adult population, replacing the least fit. The fittest n are randomly picked as parents from the adult population. The degree of negative selection pressure can be controlled by modifying the relative sizes of the two populations. Likewise the degree of positive selection pressure is set by varying n. When child and adult population sizes are equal, the algorithm resembles a canonical or generational GA. When the child population size is reduced, the algorithm resembles a steady-state GA. Typically the child population size is set to 80% of the adult size and $n = 40\%$. (For further details of this GA, refer to (Bentley, 1999)).

Unless specified, alleles are initialised randomly, with (*xp*,*yp*,*zp*) and (*x*,*y*,*z*) values between −1.0 and 1.0 and *thresh* between −10,000 and 10,000. The ranges and precision of the alleles are limited only by the storage capacity of *double* and *long* "C" data types – no range constraints were set in the code.

3.3.1. *Genetic operators*

Genes are real-coded, but genomes may comprise variable numbers of genes. Given two parent genomes, the crossover operator examines each gene of parent 1 in turn, finding the most similar gene of the same type in parent 2. Similarity is measured by calculating the differences between respective values of operator and coding regions of genes. One of the two genes is then randomly allocated to the child. If the genome of parent 2 is shorter, the child inherits the remaining genes from parent 1. If the genomes are the same length, this crossover acts as uniform crossover.

Mutation is also interesting, particularly since these genes actually code for proteins in this system. There are four main types of mutation used here:

1. Creep mutation, where (*xp*,*yp*,*zp*) and (*x*,*y*,*z*) values are incremented or decremented by a random number between 0 and 0.5, *Affinity Threshold* is incremented or decremented by a random number between 0 and 16384 and *Concentration Threshold* is incremented or decremented by a random number between 0 and 200.
2. Duplication mutation, where a (*xp*,*yp*,*zp*) or (*x*,*y*,*z*) region of one gene randomly replaces a (*xp*,*yp*,*zp*) or (*x*,*y*,*z*) of another gene. (This permits evolution to create matching promoter regions and coding regions quickly.)
3. Gene mutation, where a random gene in the genome is either removed or a duplicate added.
4. Sign flip mutation, where the sign of *Affinity Threshold* is reversed.

Crossover is always applied; all mutations occur with probability 0.01 per gene.

4. Squareroot function regression

Previous work has demonstrated how evolution can generate specific fractal proteins that interact with each other in order to produce de-

sired patterns of activation (Bentley, 2004) or to produce a specific set of commands for a robot, to guide it past obstacles (Bentley, 2003a). Here, the task is to produce the square root of a number. The input to the system is provided by setting the concentration of the first environment fractal protein (all others have a default value of 200). The output is produced by the behavioural gene(s) as described previously. Each genotype was developed ten times in succession with random input (concentration) values between 0 and 199. The fitness was the sum of the differences between the values obtained and true square-root of the input. Note that there was no fitness measure to assess the ability of this solution to survive damage – this property emerges naturally (Bentley, 2003c).

To evolve the controllers, the fractal development system was initialised with a single cell, 2 environment genes, 2 receptor gene, 2 behavioural genes and 6 regulatory genes. (With variable length genomes, evolution was free to modify these gene numbers). The operator and coding regions of the genes were randomly initialised with the alleles that defined 10 previously evolved protein fractals (Bentley, 2004). About 8 developmental steps were employed (ten times, each with a different environmental protein concentration), and the evolutionary algorithm used a population size of 100, running for 1000 generations.

All 20 runs produced outputs very close to that of the true square-root. The best was picked (see Figure 8a), the evolved fractal proteins were written into the code as constants, the genetic algorithm removed, and the resulting developmental squareroot program compiled into a stand-alone executable. (The compiler was GCC 3.3, using xcode on a Mac Powerbook G4, Mac OS 10.3.2.)

4.1. *Comparison methods*

Two other programs were used as comparisons in the experiments. The first was a fast squareroot function, written for speed of execution, provided by Hsieh.[1] This is written in C and was simply compiled to produce a stand-alone executable. The second was evolved by Landon's simple GP (Langdon, 1998).[2] This standard genetic programming engine used a function set comprising "+", "–", "*" and "/", population size of 100, max program size of 100 nodes, number of generations = 1000, probability of crossover 0.7, and mutation 0.01.

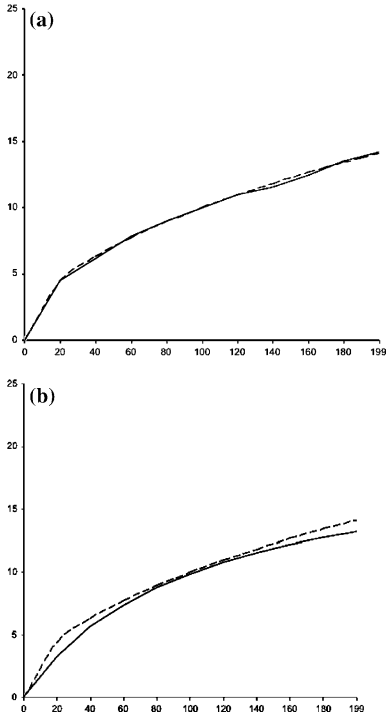

Figure 8. (a) Output of the evolved developmental squareroot function. (b) Output of the GP-evolved squareroot function. (True squareroot shown by dotted line.)

Each individual was evaluated by presenting 10 random inputs and calculating the sum of the difference between the outputs and the true squareroot of the inputs. After twenty runs, only five solutions close to the function of squareroot were evolved. Figure 8b shows the output of the best GP squareroot function. Its code is as follows (''GPdiv'' divides, trapping divide-by-zero errors):

$$
\begin{aligned}
&((\mathrm{GPdiv}(\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{GPdiv}(((\mathrm{x})-((\mathrm{x})^{*}(\mathrm{x})))-((\mathrm{x})^{*}(\mathrm{x})),(\mathrm{x})^{*}(\mathrm{x}))\\
&+\mathrm{x}+((\mathrm{x}+\mathrm{x}+\mathrm{x}0+\mathrm{x})^{*}(\mathrm{x}))-(\mathrm{x}),((\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}\\
&+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x})^{*}(\mathrm{x}))\\
&-(\mathrm{GPdiv}((\mathrm{x})^{*}(\mathrm{x}),\mathrm{GPdiv}(\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x}+\mathrm{x},(\mathrm{x})\\
&-((\mathrm{x})^{*}(\mathrm{x})))))))^{*}(\mathrm{x}));
\end{aligned}
$$

The evolved code was then isolated and compiled to produce a stand-alone GP-evolved squareroot executable.

5. Experiments and results

Although it was observed that the evolved developmental squareroot function was more accurate and this accuracy was achieved more consistently than the GP version, this was not the objective of the work. (Note that all three squareroot programs calculate the squareroot of 11 values from 0 to 199 in steps of 20, so accuracy is measured in terms of these sample points only.) Here we are more concerned with the ability of the evolved solutions to survive damage to their compiled executables. In order to assess this, a "corruption" program was written, which reads a specified file in a series of 2048 byte chunks, flipping a single, randomly chosen bit in each chunk, before saving in a new file. This was performed 50 times for all three squareroot programs, resulting in 150 corrupted executables. These were then executed and the results noted.

The initial results were perhaps predictable. Both the fast squareroot program and the GP-evolved program were approximately 16 kilobytes in size, smaller than the 28 kilobytes of the developmental squareroot program. This meant they were corrupted less, resulting in more reliable performance. Indeed, the fast squareroot program ran perfectly 15/50 times, and ran providing incorrect solutions twice. The GP-evolved program ran perfectly 10 times, and provided incorrect solutions twice. The developmental program ran perfectly twice, provided approximately correct solutions 3 times, and incorrect solutions twice.

With all three programs being different lengths and containing different code, it was clear that the comparison was flawed. A constant corruption rate per bit favours shorter programs. Also, the developmental program contained many more calls to memory-handling routines and library functions, resulting in more brittle code (which was corrupted more), and thus code more likely to fail. To overcome this, the three squareroot programs were combined into a single program. By changing a simple compiler directive, the program could be compiled in three different ways:

1. Output result of method 1 only if methods 2 and 3 executed correctly.
2. Output result of method 2 only if methods 1 and 3 executed correctly.

Table 2. Results of running 200 corrupted executables for three squareroot programmes

	Square root	GP	Ev. Dev.
Fail	197	198	177
Incorrect run	1	0	13
Graceful degradation	0	0	8
Perfect	2	2	2

Graceful degradation is defined as solutions producing 10 non-zero values within 50 percent of the correct values.

3. Output result of method 3 only if methods 1 and 2 executed correctly.

where method 1 was the fast squareroot function, method 2 was the GP-evolved function, and method 3 was the fractal developmental squareroot function. This way, all three programs contained the same code with the same susceptibility to damage, except that the code that generated the output was different in each program.

The three executables were corrupted 200 times using the method described previously. The corrupted programs were then executed and the results noted, see Table 2.

6. Analysis

The results are fascinating. Despite all three programs suffering from the same proportion and type of errors (see Figure 9), there is marked difference in performance between the developmental squareroot program and the fast squareroot and GP-evolved programs. The latter both only manage to execute correctly 2 out of 200 corrupted executables. They display zero graceful degradation – they simply fail to execute (or in a single case, execute with incorrect results).

The developmental squareroot program manages to run 23 times out of 200. About 13 of those produce incorrect results (usually all zeros). Only 2 produce perfect results (as good as the uncorrupted program). But in 8 cases, the developmental squareroot produces *approximately correct* solutions (Figure 10). The damage to the executable has perhaps corrupted the genes or fractal proteins, and the developmental program recovers. As was described in Section 2, evolution has not only evolved a good solution, it has created a solution

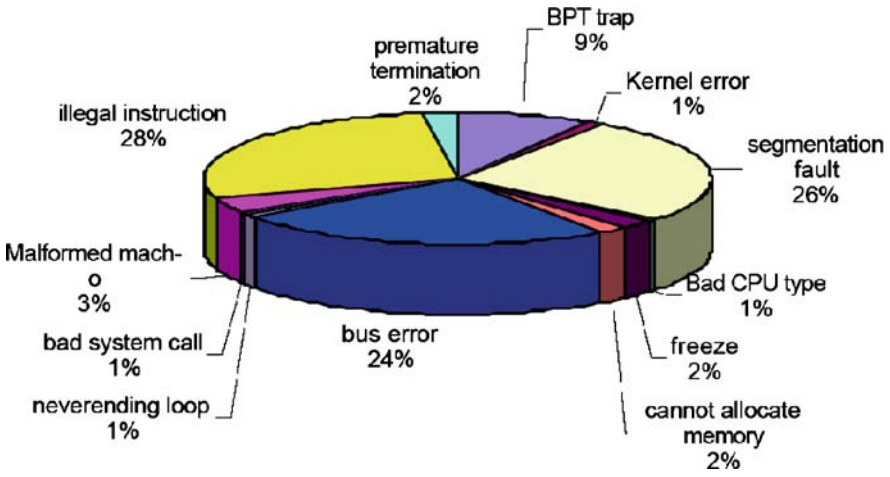

Figure 9. Percentage and type of errors obtained in all runs of the corrupted programmes.

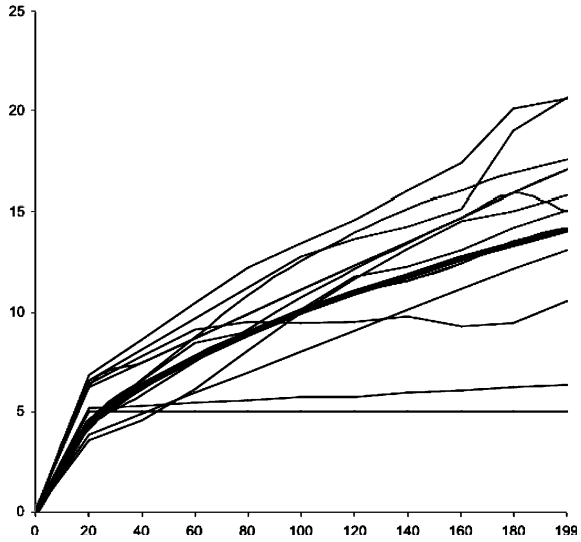

Figure 10. Outputs produced by different runs of the damaged developmental square-root function, true squareroot shown in bold. Note that most produce results that are approximately correct (within 2 of the true value), displaying remarkably graceful degradation.

that copes with damage. It seems that this protection even extends to damage done to the executable code, as well as simple mutation-driven damage to genes.

Note that the GP-evolved code does not display this property. It is conceivable that should the GP solution contain bloat (unused code), then it might survive damage more readily. However, this would not be the same phenomenon observed in the developmental system, which has no bloat (Bentley, 2003c). In the developmental system, damage to the code *that is actually being used*, can be survivable (Bentley, 2003c). It seems probable that only this form of system enables this kind of "natural" graceful degradation to emerge.

It should also be noted that all three squareroot programs were calculating the squareroot results according to their inputs. The developmental program did not, in any sense, have the answer "wired in" as constants -- indeed it performed more calculation using the input to produce the results than the other two programs. The ability to survive damage arises because of the way the calculation was performed -- the dynamic (gene) networks in the code are able to survive despite having "holes punched in them" by the corruption program.

It is conceivable that other algorithms that make use of networks in this way may also show similar tolerance to damage. Neural networks should have this property if a level of redundancy is included in their structure. Another aspect worthy of future study would be to incrementally increase damage to solutions and assess graceful degradation. At present such a study is difficult because of the extreme brittleness of operating systems and compiled code -- even a single error is usually devastating. The use of simpler systems (e.g. embedded systems) may provide useful future platforms for such studies.

7. Conclusions

Development is the process used by evolution to construct complex, adaptive and robust forms. Computer algorithms based on development can share some of these properties. Here, experiments have shown that, unlike traditional software, evolved developmental programs show graceful degradation after damage to their executable code. While surviving only around 14 bits (0.05%) of damage 10% of the time is not a great achievement compared to the robustness of natural systems, given the conventional (brittle) nature of the programming language, compiler and hardware, it is still considered impressive. It seems likely that should computer science remove its brittleness and embrace evolutionary and developmental systems more fully, abilities such as graceful degradation will improve further.

Acknowledgements

Thanks to the anonymous reviewers for their helpful comments. This material is based upon work supported by the European Office of Aerospace Research and Development (EOARD), Airforce Office of Scientific Research, Airforce Research Laboratory, under Contract No. F61775-02-WE014. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of EOARD. MOBIUS is an EMBLEM project.

Appendix

In this Appendix we describe in detail the development algorithm for mapping genotype to phenotype. The genetic algorithm (described in Section 3.3) calls this function to evaluate evolving individuals. Phenotype fitness may be measured during development by monitoring the change in output produced by behavioural genes. For the square-root problem, the final output after 10 developmental steps is used; this is developed ten times in succession using different concentrations of environmental proteins as inputs and compared with 10 correct exemplars.

Create the cell

Examine the type field of all genes and extract all: environment, receptor, regulatory and behavioural genes

Calculate and store all fractal proteins defined by cis-sites and coding regions of all genes

Fill cytoplasm with all possible fractal proteins defined by coding regions with concentration of zero

Loop

{

Set concentration of environment protein to input value of squareroot function.

Mask environment proteins with receptor fractals and merge, calculate concentrations and add to cytoplasm.

Merge cytoplasm fractals (ignore those with concentration of zero).

Calculate merged concentration.

Compare merged cytoplasm fractal with cis-sites of all behavioural and regulatory genes

If the affinity threshold of the current gene is +ve
Activate gene based on activation probability (a function of the cytoplasm-cis-site match), see Equation 3
If the gene is behavioural
Increase output by a function of concentration and fate (Equation 5)
If the gene is regulatory
Concentration of protein defined by gene coding region will be increased by a function of input concentration on cis-site (Equation 2)
else if the affinity threshold of the current gene is −ve
Activate gene based on inverse of activation probability (a function of the cytoplasm-cis-site match), see Equation 3
If the gene is behavioural
Decrease output by a function of concentration and fate (Equation 4)
If the gene is regulatory
Concentration of protein defined by gene coding region will be increased by a function of input concentration on cis-site (Equation 2)
Update concentration of all cytoplasm proteins, increasing by amount defined by genes (Equation 2), decreasing by decay level (Equation 1).
} until desired number of iterations *d*.

Notes

[1] http://www.azillionmonkeys.com/qed/sqroot.html#fast
[2] http://www.cs.ucl.ac.uk/staff/w.langdon/ftp/gp-code/simple/simple-gp.c

References

Bentley PJ (2004) Fractal proteins. 2004. Genetic Programming and Evolvable Machines.

Bentley PJ (2003a) Evolving fractal gene regulatory networks for robot control. In Proceedings of ECAL 2003.

Bentley PJ (2003b) Evolving fractal proteins. In Proceedings of ICES '03, the 5th International Conference on Evolvable Systems: From Biology to Hardware.

Bentley PJ (2003c) Evolving beyond perfection: an investigation of the effects of long-term evolution on fractal gene regulatory networks. In Proceedings of Information Processing in Cells and Tissues (IPCAT 2003).

Bentley PJ (1999) From coffee tables to hospitals: generic evolutionary design. In: Bentley PJ (ed.), Evolutionary Design by Computers., pp. 405–423. Morgan Kaufmann Pub, San Francisco Chapter 18

Jackson AH and Tyrrell AM (2002) Implementing asynchronous embryonic circuits using AARDVArc. In Proceedings of 2002 NASA/DoD Conference on Evolvable Hardware (EH-2002), IEEE Computing Society, Alexandria, Virginia, pp. 231–240

Kumar S and Bentley PJ (2003) Computational embryology: Past, present and future. In:Ghosh and Tsutsui (eds) Theory and Application of Evolutionary Computation: Recent Trends, Springer Verlag, UK

Langdon W (1998) Genetic Programming + Data Structures = Automatic Programming. Kluwer Pub

Mahdavi S and Bentley PJ (2003) Adaptive evolutionary motion of smart robots. In Proceedings of EvoROB 2003, 2nd European Workshop on Evolutionary Robotics.

Mandelbrot B (1982) The Fractal Geometry of Nature. W.H. Freeman and Company

Miller J and Banzhaf, W (2003) Evolving the program for a cell: from French flags to Boolean circuits. In: Kumar S and Bentley PJ (eds) On Growth, Form and Computers. Academic Press

Shelton C and Koopman P (2001) Developing a software architecture for graceful degradation in an elevator control system. Workshop on Reliability in Embedded Systems

Thompson, A (1997) Evolving inherently fault-tolerant systems. Proc. Instn. Mech. Engrs.

Wolpert L, Beddington R, Jessell T, Lawrence P, Meyerowitz E and Smith J (2001) Principles of Development, 2nd edn. Oxford University Press