Controlling Robots with Fractal Gene Regulatory Networks

Peter J. Bentley

Department of Computer Science, University College London, Gower Street, London.

P.Bentley@cs.ucl.ac.uk

Abstract. Fractal proteins are a new evolvable method of mapping genotype to phenotype through a developmental process, where genes are expressed into proteins comprised of subsets of the Mandelbrot Set. The resulting network of gene and protein interactions can be designed by evolution to produce specific patterns, that in turn can be used to solve problems. This chapter introduces the fractal development algorithm in detail and describes the use of fractal gene regulatory networks for learning a robot path through a series of obstacles. The results indicate the ability of this system to learn regularities in solutions and automatically create and use modules.

Keywords: evolutionary computation, computational development, robot control, gene regulatory networks, fractal proteins

1 Introduction

Life is complex. This is true in the chemical interactions of proteins and genes within a single cell, or in the cellular interactions in a multicellular organism. While evolution is mostly to blame for this, there can be little doubt that complexity could not arise if the vast intricacies of molecular interactions and physical forces were not present. Open-ended evolution (evolution in which solutions get progressively more complex) relies on the right kind of genetic representation in the right kind of environment. In nature, this is DNA – a molecule that relies on chemical interactions in order to function.

Translating these ideas into computer science is not a simple prospect. As we know in evolutionary computation, using a fixed binary string as a genotype, prohibits complexity growth. But variable-length representations such as genetic programming do not guarantee an increase in complexity either (unless you count introns as complexity). Even if a seemingly ideal representation is found, often it is not evolvable, or it only achieves its complexity increases through the careful high-level structures (e.g. modularity) imposed on it by the developer.

This work takes a different approach. A developmental process maps variable-length genotypes to phenotypes, through the use of *fractal proteins*. Genes are expressed into complex fractal shapes (subsets of the Mandelbrot Set) that interact according to their forms. The resulting network of gene interactions can be designed by evolution to produce specific

gene activation patterns, that in turn can be used to solve problems. In this chapter the use of fractal gene regulatory networks for learning a robot path through a series of obstacles is described.

2 Background

2.1 Development

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 "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). An introduction to biological development can be found in (Bentley, 2002), with greater detail in (Wolpert et al, 2001) and (Kumar and Bentley, 2003a).

There is currently much research being performed on computational development. Problems of scalability, adaptability and evolvability have led many researchers to attempt to include processes such as growth, morphogenesis or differentiation in their evolutionary systems (Jackson and Tyrrell 2002, Sipper 2002, Miller and Banzhaf 2003). However, research focussing on the creation of GRNs are less common in Computer Science. Jakobi (1995) was one of the first to explicitly design a system which enabled (and even extracted) GRNs, which were then used to develop neural networks for controlling robots. Jakobi's genetic representation was highly flexible, but had no real concept of chemistry to affect his proteins. While Jakobi was enthused by the results at the time, the subsequent lack of further development implies that his proposed representations were not as evolvable as first thought. Another example is the work of Eggenberger (1997), which still remains impressive, demonstrating the use of gene behaviours such as regulation, and cellular behaviours such as differentiation and simple morphogenesis. Eggenberger modelled biological development in some detail, and showed how different forms made from cell clusters could be evolved using these techniques. Nevertheless, Eggenberger's work has little concept of gene expression or protein folding, and the evolvability and scalability of his system remains open to question. More recently, Bongard (2002) continues research that follows many of Eggenberger's ideas. He describes the use of GRNs to generate morphologies and neural networks of simple robots within a physics simulator. Again his work captures many features of biological development, and even shows the spontaneous creation of modularity within genotypes. However his "GRNs" are unusual in that they often do not appear to act in autocatalytic loops (i.e., they seem to be gene regulatory trees rather than networks), removing many of the advantages gained by more normal "looping" GRNs. Other examples include the work of Kennedy (1999) and the early work of Rosenberg (1967).

Researchers such as Hornby (2003), Bongard (2002), and Kumar and Bentley (2003) have demonstrated that various types of development can enable smaller genotypes to represent more complex phenotypes through the ability of development to discover modularities and

repetition. Other scientists in the field have been focussing on the ability of developmental methods to enable self-repairing behaviour and graceful degradation of solutions. For example, the work of Andy Tyrrell and his group create fault-tolerant hardware inspired by ideas of embryology and immune systems (Jackson and Tyrrell, 2002). This is being performed in collaboration with Moshe Sipper (2002) who has demonstrated a simple self-repair capability for electronic circuits by the use of cellular-automata-like "biodules". Using ideas inspired from embryology, the circuits can functionally self-organise, while redundant biodules enable damage to be overcome. Adrian Thompson has spent some years investigating how evolvable hardware can provide robust solutions, for example circuits that handle large variations in temperature and fabrication, by testing designs in different environments during evolution (Thompson and Layzell, 2000). 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. 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). However, work on applying developmental algorithms to robot control is less common. Most seem to rely on the development of neural networks that are then used to control motion (and in some cases generate form) (Jakobi 1995; Hornby, 2003; Bongard 2002).

Research performed in collaboration with the author has been focusing on development for some years (e.g. see (Kumar and Bentley, 2003a,b)). Kumar, (with Wolpert) has developed extensive modelling software capable of evolving genes that are expressed into proteins, forming GRNs that control the expression of the genes and the development (growth, placement, function) of cells (Kumar and Bentley 2003c). Gordon has been investigating the use of development-inspired evolutionary systems for evolvable hardware, which makes use of ideas such as differentiation and growth (Gordon and Bentley 2002).

The research described here complements these systems, by focusing on methods to enrich the genetic space in which evolution searches. Specifically, this paper introduces the novel idea of *fractal proteins*. Here, a gene defines a protein constructed from a subset of the Mandelbrot Set. This can be contrasted with the simple artificial chemistries by Bagley, Farmer and others (Bagley et al, 1991, Farmer et al 1986). As will be shown, fractal proteins enable substantially more complex interactions than simple string-matching systems.

2.2 Mandelbrot Set

Given the equation $x_{t+1} = x_t^2 + c$ where x_t and c are imaginary numbers, Benoit Mandelbrot wanted to know which values of c would make the length of the imaginary number stored in x_t stop growing when the equation was applied for an infinite number of times. He discovered that if the length ever went above 2, then it was unbounded – it would grow forever. But for the right imaginary values of c, sometimes the result would simply oscillate between different lengths less than 2.

Mandelbrot used his computer to apply the equation many times for different values of c. For each value of c, the computer would stop early if the length of the imaginary number in x_t

was 2 or more. If the computer hadn't stopped early for that value of c, a black dot was drawn. The dot was placed at coordinate (m, n) using the numbers from the value of c: $(m + ni)$ where m was varied from –2.4 to 1.34 and n was varied from 1.4 to -1.4, to fill the computer screen. The result was the infinite complexity of the "squashed bug" shape we know so well today (Mandelbrot, 1982).

3 Fractal Proteins

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. The motivations behind this work can be listed as follows:

1. Natural evolution extensively exploits the complexity, redundancy and richness of chemical systems in the design of DNA and the resulting developmental systems in organisms. Providing a computer system with genes that define fractal proteins gives the system complexity, redundancy and richness to exploit.

2. It is extremely difficult and computationally intensive to model natural chemical systems accurately in an artificial chemistry. Fractal proteins have many of the same properties as natural proteins, without any modelling overheads.

3. A fractal protein (with the infinite complexity of the Mandelbrot set) can be defined by just three genes.

4. The "fractal genetic space" is highly evolvable – a small change to a gene produces a small change to the fractal protein, while the self-similarity of fractals ensures that any specific shape can be found in an infinite number of places.

5. When fractal proteins are permitted to interact according to their morphologies, a hugely complex (and eminently exploitable) fractal chemistry emerges naturally.

6. Calculating subsets of Mandelbrot sets is *fast* so there is little overhead.

Further motivations and discussions on fractal proteins are provided in (Bentley, 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.

Table 1. Types of objects in the representation

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 (or promoter) 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	structural gene comprising operator and cellular behaviour region.

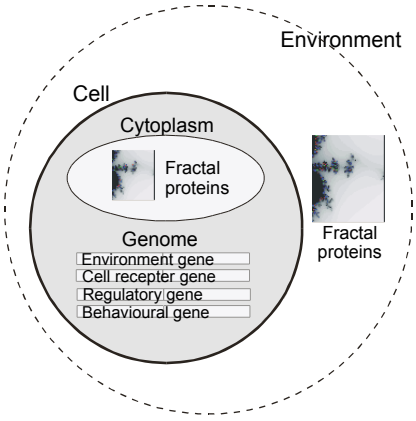

Fig. 1. Representation using fractal proteins.

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.

Fig. 2. The fractal development algorithm.

It should be noted that the system used for the experiments in this chapter is an extended version of systems described elsewhere. While the detail provided below is in some respects a distraction from the theme of this paper, for reasons of reproducibility it is provided in full.

3.1 Defining a Fractal Protein

In more detail, a fractal protein is a finite square subset of the Mandelbrot set, 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 *z*, see fig. 3 for an example. In this way, it is possible to achieve as much complexity (or more) compared to natural protein folding in nature.

In addition to shape, each fractal protein represents a certain *concentration* of protein (from 0 meaning "does not exist" to 200 meaning "saturated"), determined by protein production and diffusion rates.

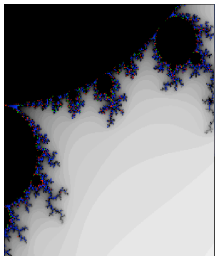

Fig. 3. Example of a fractal protein defined by (*x*=0.132541887, *y*=0.698126164, *z*=0.468306528)

3.2 Fractal Chemistry

Cell cytoplasms and the environment usually contain more than one fractal protein. In an attempt to harness the complexity available from these fractals, multiple proteins are merged. 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). Intuitively, this results in black regions being treated as though they are transparent, and paler regions "winning" over darker regions. See fig 4 for an example.

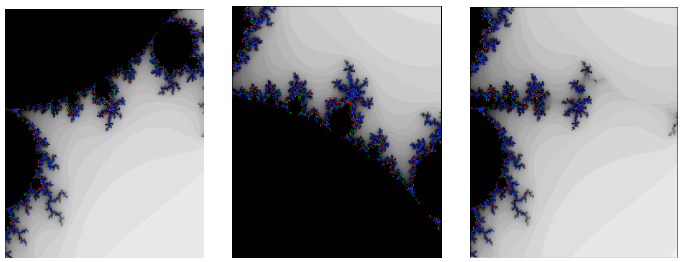

Fig. 4. Two fractal proteins (left and middle) and the resulting merged fractal protein combination (right).

3.3 Calculating concentration levels

The total concentration of two or more merged fractal proteins is the mean of the different concentrations seen in their merged product. For example, fig. 4 shows how fractal proteins are merged to form a new fractal shape. Figure 5 illustrates the resultant areas of different concentration in the product. When being compared to the (*xp, yp, zp*) promoter region of a gene (the "conditional" part of the gene to be matched, see later section on genes), the concentration seen on that promoter is described by all those regions that "fall under" the promoter, see fig. 5. 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, see fig. 6.

Fig. 5. The different concentrations of the two fractal proteins (left and middle) and the concentration levels in their merged product (right).

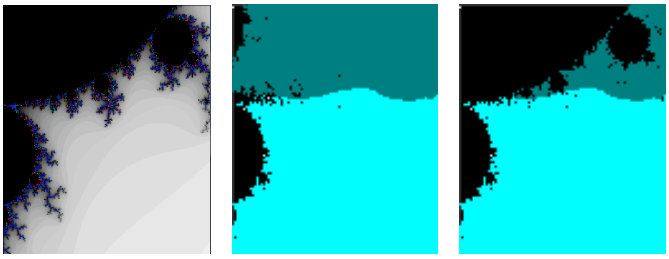

Fig. 6. The shape of the desired protein as defined by a promoter (left), the shape and concentration levels of merged proteins in the cytoplasm (middle) and the concentration levels seen on that promoter (right), where total concentration is taken as mean. Note that although a merged protein may decrease affinity (similarity) to the promoter, should the second protein have a higher concentration level to the first, it will boost overall concentration seen by the promoter, i.e., act like a catalyst to speed up (or slow down, if lower) the "reaction".

3.4 Updating Protein Concentration Levels

Every time step the new concentration of each protein is calculated. This is formed by summing two separate terms: the previous concentration level after diffusion (*Dc*) and the new concentration output by a gene (*Gc*). These two terms model the reduction in concentration of proteins over time, and the production of new proteins over time, respectively, where:

$$Dc = Pc - Pc / C_p + 0.2 \qquad \textbf{Equation 1}$$

Pc is protein concentration in previous time step,
C_p is a constant normally set to 5;
the final addition of 0.2 ensures a minimum level of diffusion

and:

$Gc = Tc \times Cm$,

Tc is the mean concentration seen at the promoter,
Cm is a concentration multiplier, where:

$$Cm = \tanh((Tc - ct) / C_w) / C_i \qquad \textbf{Equation 2}$$

where: *ct* is the concentration threshold from the gene promoter
C_w is a constant (set to 30 for these experiments)
C_i is a constant (set to 2 for these experiments)

The value of C_p determines the rate of protein decay, and was set to provide a useful level. (Increasing the value will reduce protein decay, extending the time proteins remain in the cytoplasm. This might make longer timings possible, but could reduce control of short-term concentration changes. Reducing the value will make proteins decay quickly, providing better short-term control, but making long-term effects harder to achieve.) The values of C_w and C_i determine the scaling of *Gc* with respect to *Tc* – the lower the values, the lower the rate of increase of *Gc* as *Tc* increases.

Should a gene be activated (the affinity matches above/below the given threshold, and concentration above zero, see earlier), it will produce the resultant protein. To do this, the concentration of that protein is modified by incrementing with the result of a function of the concentration threshold (*ct*) and the mean total concentration seen at the promoter (*Tc*), given as Equation 2. 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 *ct* threshold) will increase the diffusion rate of the output protein.

3.5 Genes

The environment gene, cell receptor gene, regulatory genes, and behavioural genes all contain 7 real-coded values:

xp	*yp*	*zp*	*Affinity threshold*	*Concentration threshold*	*x*	*y*	*z*	*type*

where (*xp, yp, zp, Affinity threshold, Concentration threshold*) defines the promoter (operator or precondition) for the gene and (*x,y,z*) defines the coding region of the gene. The *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 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 *Pa* of a gene is given by:

$$Pa = (1 + \tanh((m - At - C_t) / C_s)) / 2 \qquad \textbf{Equation 3}$$

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

Note that the values of C_t and C_s were chosen with respect to the range of values of *m* (which depend on the sampling resolution used, but in the experiments varied between 0 and around 10000) to provide a small, but distinct transitional region between probabilities. Using smaller values of C_t and C_s makes the transition from probability 0 to 1 more discrete; larger values makes the transition more gradual. With the values chosen, a small amount of noise is produced (observed during tests in which the same calculation was repeatedly performed). This noise is seen as actively beneficial to evolution in this context, enabling it to sample the effects of alternative activations occasionally as it fine-tunes values for *At.*

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 (*ct*) and the mean total concentration seen at the gene promoter (*totalconc*), as given earlier. 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

ct 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.

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, where all black regions of the mask are treated as opaque, and all other regions treated as transparent. For an example, see fig. 7. This means that the more similar a protein or protein compound is to the receptor, the more likely it will be permitted into the cytoplasm intact. This coincides nicely with cell receptors in nature, which are under genetic control and selectively permit specific proteins through the cell wall into the cytoplasm.

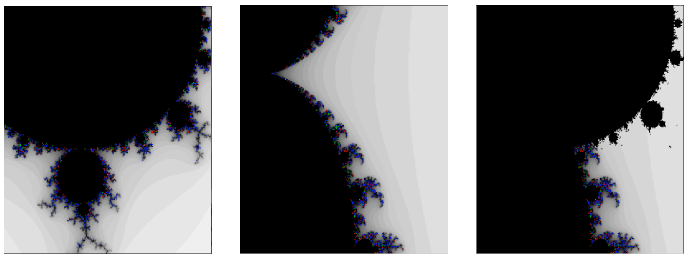

Fig. 7. Cell receptor protein (left), environment protein (middle), resulting masked protein to be combined with cytoplasm (right).

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.

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, calculated as follows:

If the gene is being activated with a negative *Affinity threshold*,
output = *output* _ (*totalconcentration* _ *concentrationthreshold*) _ *fate*

If the gene is being activated with a positive *Affinity threshold*,
output = *output* + (*totalconcentration* _ *concentrationthreshold*) _ *fate*

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 are more behavioural genes than are required, the resultant behaviour will be the sum behaviour of all genes.)

3.6 Fractal Sampling

All fractal calculations (masking, merging, comparisons) are performed at the same time, by sampling the fractals at a resolution of 15x15 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 being compared 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.

3.7 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. 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.

3.8 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 -10000 and 10000. 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.

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 parent1 in turn, finding the most similar gene of the same type in parent 2. Similarity is measured by calculating the differences between values of operator and coding regions of genes. One of the two genes is then randomly allocated to the child. If the genome of parent2 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 Robot Control

Previous work has demonstrated how evolution can generate specific fractal proteins that interact with each other in order to produce desired patterns of behavioural gene activation (Bentley 2003a,b,c). Here the two behavioural genes are used to generate commands for a robot, with the aim of directing the robot past obstacles in its environment and reach a destination. So instead of directing cellular behaviour (i.e., cell division, differentiation or death), the fractal gene regulatory networks will direct robot behaviour. Although the tasks are clearly very different, both do rely on precise timing and accuracy, and both make use of the inherent pattern-generating properties of GRNs.

4.1 Robot

The platform used was a palm-sized six-legged robot known as a *wonderborg*™, produced by Bandai, see Fig. 8. Although not widely available outside Japan, the robot boasts several useful features: it is programmed via infrared communications, and once programmed it runs fully autonomously. It has two forward-facing infrared collision detectors, a bottom-facing infrared floor sensor, an upwards facing light sensor and two microswitch touch sensors or feelers. Its legs are driven by two motors (one for the legs on each side), operating like a tracked vehicle, and allowing it to walk where Khepera robots cannot.

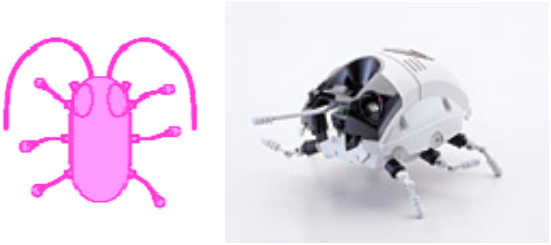

Fig. 8. The *wonderborg*™ robot (Bandai) has six legs, two feelers, two infrared collision detection sensors, a floor sensor, a light sensor and capabilities for the addition of a further motor and sensor.

4.2 Mapping Genes to Movement

The *wonderborg*™ was not designed as a research platform, so unfortunately suffers from certain drawbacks. Currently the only way to send a program to the robot is through Bandai's proprietary software which forces the use of high-level commands and programming structures. In some cases these commands are useful, for example those that send specific infrared signals or detect specific signals (for use in inter-robot communications). But all movement commands are also high-level (e.g. "Rotate right" or "Back up left"), prohibiting the independent control of motors. Although a translation from independent motor commands to the high-level commands is theoretically possible, the proprietary software also has a fairly small capacity for storing the commands, limiting this option. To overcome these difficulties, the two behavioural genes were treated as "steering" and "acceleration," and then their values were mapped onto the appropriate high-level commands. For example, a positive value for the steering gene is taken to mean "steer right" and a value of -3 for the acceleration gene is taken to mean "move -3 backwards" (move back three steps), see table 2. The fractal development system was then modified to convert the patterns of gene activation produced during development into a robot control file in the appropriate format for the proprietary software.

Table 2. Values of the steering and acceleration genes are mapped to the nine predefined robot commands

Steering		Acceleration		Robot command	Symbol (*n*=1)
0	(None)	0	(None)	*Halt*	
-ve	(Left)	0	(None)	*Rotate left*	
+ve	(Right)	0	(None)	*Rotate right*	
0	(None)	*n*	(*n* forwards)	*Advance* by *n*	
-ve	(Left)	*n*	(*n* forwards)	*Turn left* by *n*	
+ve	(Right)	*n*	(*n* forwards)	*Turn right* by *n*	
0	(None)	-*n*	(*n* backwards)	*Back up* by *n*	
-ve	(Left)	-*n*	(*n* backwards)	*Back up left* by *n*	
+ve	(Right)	-*n*	(*n* backwards)	*Back up right* by *n*	

4.3 Simulator

Finally, to enable high-speed evaluation of robot control programs, a *wonderborg*™ simulator was created. This reads the same file format as used by Bandai's proprietary

software (and as output by the fractal developmental system) and calculates the path of the robot through an environment. Because the *wonderborg*™ has various interchangeable legs, the simulator can be calibrated (in terms of noise and movement) for different setups. (Note that some of the legs make control almost impossible, with noise of 10% in distance travelled and 20% in rotational angle per command. The experiments used the simulator calibrated for a more reliable setup.) The simulator was designed to be fast – on a 1 Ghz PC, approximately 40 developmental cycles and corresponding robot control simulations occur every second.

5 Experiments

Two sets of experiments were performed. The first used a simple environment with four obstacles, the second used a more difficult environment with seven obstacles, see figure 9. The robot simulator was initialised with the robot at one end of the environment. The further the robot managed to walk across the environment the higher the fitness of the corresponding controller. If the robot touched an obstacle or walked off the edges, its final position was measured as its last valid position in the environment. Note that the controllers did not use any of the robot's sensors to gauge proximity of obstacles (which might have made the task easier). Instead they specified a fixed path past the obstacles, learning the structure of the environment through generations of "hitting its head against walls." A second fitness measure (of less importance than the first) was used to provide penalties corresponding to the time taken by the robot and hence encourage efficient and fast journeys. (Without this, the robots would often rotate many times on one spot for several seconds between each movement in order to find the perfect angle.)

To evolve the controllers, the fractal development system was initialised with a single cell, 1 environment gene, 1 receptor gene, 2 behavioural genes and 6 regulatory genes. (Note that 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, 2003a). 32 developmental steps were employed, and the evolutionary algorithm used a population size of 100, running for up to 500 generations in the first experiment and 1500 in the second.

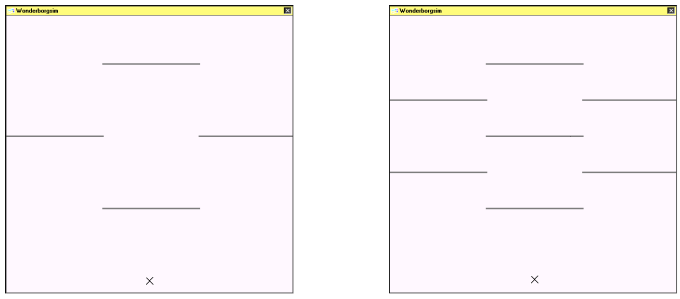

Fig. 9. The two environments used for the experiments. The starting position of the robot is marked with an "X." The higher (in the window) the robot manages to walk, the fitter the controller.

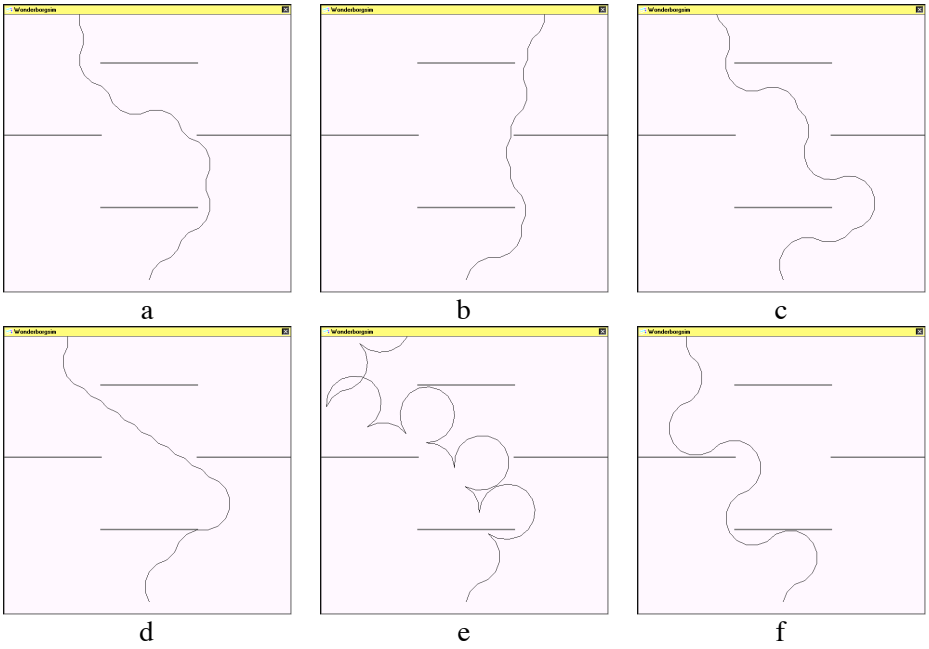

Fig. 10. Six examples of very fit controllers for the first experiment.

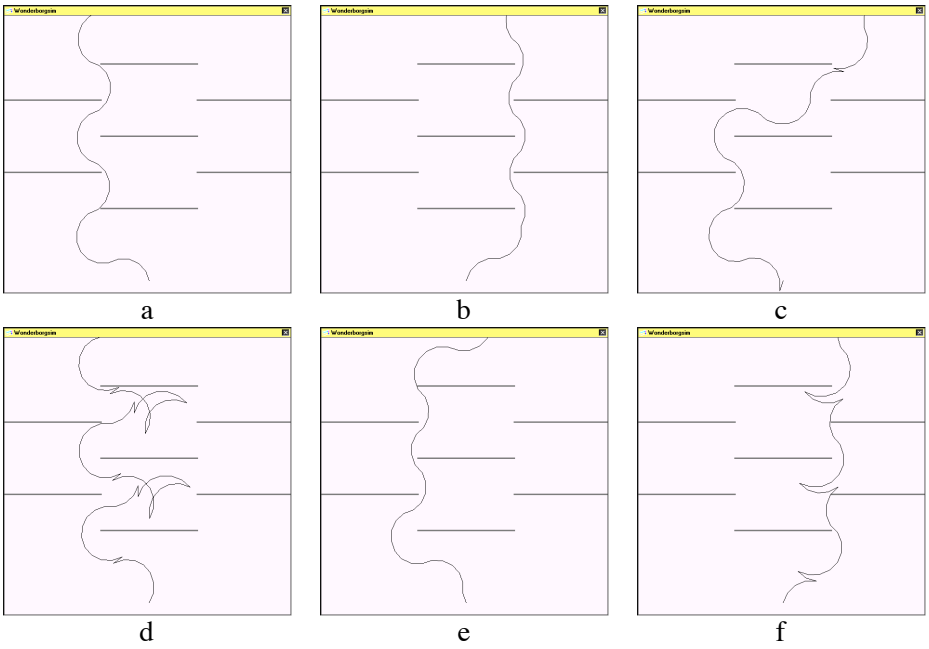

Fig. 11. Six examples of very fit controllers for the second experiment.

6 Results and Analysis

In the first experiment, 13 out of 20 produced robot controllers able to reach the top (500 generations). In the second experiment, only 1 out of 20 produced successful robot controllers after 500 generations. When the GA was allowed to run for 1500 generations, 11 out of 20 produced robot controllers able to reach the top. The increase in success is further evidence of evolvability of this representation (Bentley 2003a). Figs 10 and 11 show examples of robot paths produced by fit controllers. The final controllers have been confirmed by testing on the actual robot.

The results were surprisingly good. Despite the poor level of control available through the high-level commands, evolution and development manage to create remarkably precise and detailed robot paths, designed to avoid the obstacles in the environment. Significantly, these controllers often employed "modules" or "subroutines" that were repeated (sometimes with variations) to overcome the obstacles. This automatic discovery and exploitation of pattern in the solution is very important, and also very unusual – most other systems (e.g. ADFs in GP or parameterised L-Systems) must explicitly build in notions of modules. Here they emerge naturally. To illustrate a little of how this happens, figure 12 displays protein concentrations during development of the controller in fig 11c.

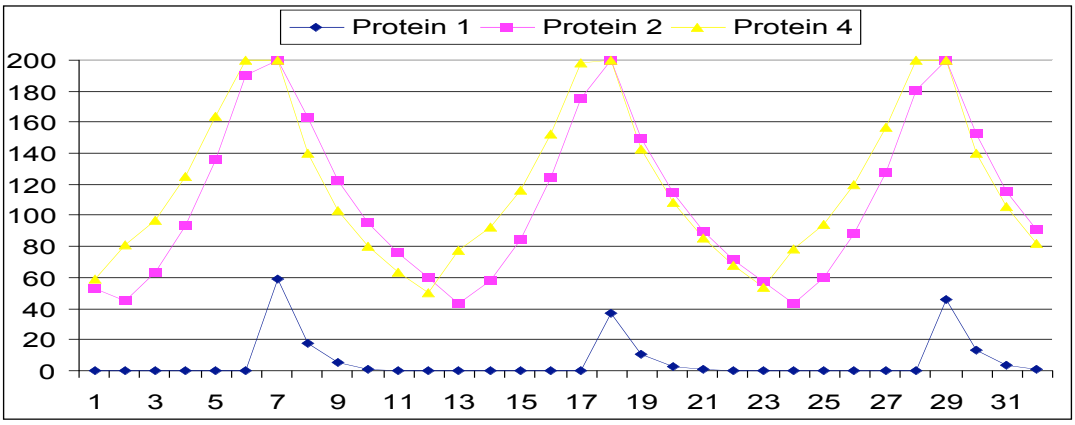

Fig. 12. Protein concentrations of the three main proteins used in controller shown in fig 11c. Note how steering and acceleration are the result of interactions between oscillating protein concentrations, resulting in very precisely controlled oscillating robot motion. The 32-step pattern is partially repeated in the final controller, producing a total of 50 commands for the robot.

Fig. 13. The 32-step controller corresponding to the protein concentrations in Fig. 12 (read left to right). Note that the sequence partially repeats (up to command 18) before the robot reaches its destination.

In this solution, 50 commands are executed. The final genome comprises thirteen genes, out of which the following are used: three regulatory genes producing three proteins, three behavioural genes, one environment gene and three receptor genes. Figure 13 shows the actual commands sent to the robot. Although this controller does not have a single, clearly repeated "module" such as those evident in Fig. 11d and f, it has discovered that a "subroutine" of turning left, followed by a "subroutine" of turning right is important. Underlying the commands are repeated sequences of gene activations. These movements are repeated until the robot reaches its destination, but the *degree* of the movements is subtly modified in each repetition by the concentrations within the GRN, ensuring that the final movement is asymmetrical and the robot avoids all obstacles.

Finally, it is clear that this system, although allowing variable genomes, does not behave in the same manner as GP systems. Previous work has shown that genomes are just as likely to remain constant or decrease in length as they are to increase in length (Bentley 2003c).

7 Conclusions

It is not a trivial task to find a suitable genetic representation that enables open-ended evolution, while being evolvable, incorporating ideas of intricate chemical/physical environments, and employing developmental processes. The use of genes expressed as fractal proteins is the approach investigated in this paper. The work has shown that fractal gene regulatory networks can be successfully designed by evolution to solve problems. Here, the task of producing robot controllers capable of guiding a "bug" robot past obstacles in an environment was demonstrated. In addition to creating diverse and useful controllers, the fractal GRN also demonstrated an ability to identify patterns in solutions and create its own modules (subsolutions that were reused and used with minor modifications). This automatic learning, not just of a good solution (robot path), but of a way of *building* a good solution in an efficient manner, is seen as a significant and important property of developmental processes. Combine this with the results of previous research that shows a tendency towards efficiency and fault-tolerance of fractal GRNs (Bentley 2003c), and the potential for this technique looks impressive.

Acknowledgments

This chapter is an extended version of the paper "Evolving Fractal Gene Regulatory Networks for Robot Control" presented at ECAL 2003. Thanks to Sanjeev Kumar for his 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.

References

R. J. Bagley, J. D. Farmer, and W. Fontana, (1991) Evolution of a Metabolism. In *Artificial Life II*, Langton, C. G., Farmer, J. D., and Rasmussen, S., (eds), Addison-Wesley, 1991, pp. 141-158.

Bentley, P. J. (2003a) Fractal Proteins. To appear in *Genetic Programming and Evolvable Machines Journal*. Kluwer Pub.

Bentley, P. J. (2003b). Evolving Fractal Proteins. In *Proc. of ICES '03, the 5th International Conference on Evolvable Systems: From Biology to Hardware*. Springer Verlag.

Bentley, P. J. (2003c) Evolving Beyond Perfection: An Investigation of the Effects of Long-Term Evolution on Fractal Gene Regulatory Networks. In Proc of *Information Processing in Cells and Tissues* (IPCAT 2003).

Bentley, P. J. (2002) *Digital Biology*. Simon and Schuster, NY.

Bentley, P. J. (1999) From Coffee Tables to Hospitals: Generic Evolutionary Design. Chapter 18 in Bentley, P. J. (Ed) *Evolutionary Design by Computers*. Morgan Kaufmann Pub. San Francisco, pp. 405-423.

Bongard, J. C. (2002) Evolving Modular Genetic Regulatory Networks. In *Proceedings of The IEEE 2002 Congress on Evolutionary Computation (CEC2002)*, IEEE Press, pp. 1872-1877.

Eggenberger, Peter. (1997) Evolving Morphologies of Simulated 3d Organisms Based on Differential Gene Expression. 1997. In: Husbands, P. & Harvey, I. (eds.) *Proceedings of the 4th European Conference on Artificial Life (ECAL97)*. Cambridge: MIT Press. ftp://ftp.ifi.unizh.ch/pub/institute/ailab/techreports/97.20.ps.gz

Eigen, M. (1987) New concepts for dealing with the evolution of nucleis acids. In *Cold Spring Harbor Symposium on Quantitative Biology*, vol LII.

J. D Farmer, N. H. Packard, and A. S. Perelson (1986) "The Immune System, Adaptation, and Machine Learning", *Physica 22D*, 187-204.

T. W. Gordon and P. J. Bentley (2002). Towards Development in Evolvable Hardware. In *Proc. of the 2002 NASA/DoD Conference on Evolvable Hardware* (EH-2002, Washington D.C., July 15-18, 2002), Stoica, A., Lohn, J. and Katz, R. (Eds), IEEE Press. 2002, pp. 241-250.

P. C. Haddow, G. Tufte, and P. van Remortel. (2001) Shrinking the Genotype: L-Systems for EHW. In *Proc. Of 4th Int. Conf. On Evolvable Systems: From Biology to Hardware*, Tokyo, Japan.

Hornby, G. S. (2003) *Generative Representations for Evolutionary Design Automation*. Brandeis University, Dept. of Computer Science, Ph.D. Dissertation.

A.H. Jackson, A.M. Tyrrell (2002) Implementing Asynchronous Embryonic Circuits using AARDVArc. 2002. In *Proceedings of 2002 NASA/DoD Conference on Evolvable Hardware* (EH-2002), IEEE Computing Society, Alexandria, Virginia, Pages 231-240, 15-18.

N. Jakobi. (1995). Harnessing Morphogenesis. *International Conference on Information Processing in Cells and Tissues*, Liverpool, UK.

P. J. Kennedy (1999) *Simulation of the Evolution of Single Celled Organisms with Genome, Metabolism and Time-Varying Phenotype*, PhD Thesis, University of Technology, Sydney, 1999.

S. Kumar and P. J. Bentley. (Eds) (2003a). *On Growth, Form and Computers*. Academic Press, London.

S. Kumar and P. J. Bentley (2003b). Computational Embryology: Past, Present and Future. Invited chapter in Ghosh and Tsutsui (Eds) *Theory and Application of Evolutionary Computation: Recent Trends.* Springer Verlag (UK).

S. Kumar and P. J. Bentley (2003c) "Biologically Plausible Evolutionary Development." In *Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware* (ICES 2003). Tyrrell, A., , Haddow, P. and Torresen, J. (eds). Springer LNCS 2606, pp. 57-68.

Linden, D. (2002) Innovative Antenna Design using Genetic Algorithms. Chapter 20 in Bentley, P. J. and Corne, D. W. (Eds) *Creative Evolutionary Systems*. Morgan Kaufmann Pub, San Francisco, pp. 487-510.

Mandelbrot, B. (1982) *The Fractal Geometry of Nature*. W.H. Freeman & Company.

Mahdavi S. and Bentley P. J. (2003) Adaptive Evolutionary Motion of Smart Robots. In Proc of EvoROB2003, *2nd European Workshop on Evolutionary Robotics*, pp. 655-664.

Miller, J. and Banzhaf, W. (2003). Evolving the Program for a Cell: From French Flags to Boolean Circuits. 2003. Invited chapter in Kumar, S. and Bentley, P. J. (Eds) *On Growth, Form and Computers.* Academic Press, 2003.

T. Quick, K. Dautenhahn, C. Nehaniv, and G. Roberts. (1999). The Essence of Embodiment: A framework for understanding and exploiting structural coupling between system and environment. 1999. In *Proc. Of Third Int. Conf. On Computing Anticipatory Systems* (CASYS'99), Symposium 4 on Anticipatory, Control and Robotic Systems, Liege, Belgium, pp. 16-17.

R. S. Rosenberg, (1967) *Simulation of Genetic Populations with Biochemical Properties*, Ph.D. thesis, University of Michigan.

Sipper, M. (2002) *Machine Nature: The Coming of Age of Bio-Inspired Computing*. 2002. McGraw-Hill, New York.

Thompson, A. and Layzell, P. (2000) Evolution of Robustness in an Electronics Design. In *3rd Int. Conf. On Evolvable Systems* (ICES2000): From Biology to Hardware, p.218-228.

Lewis Wolpert, Rosa Beddington, Thomas Jessell, Peter Lawrence, Elliot Meyerowitz, Jim Smith. (2001) *Principles of Development, 2nd Ed*. Oxford University Press.