

Finding Approximate Analytic Solutions to Differential Equations Using Genetic Programming

Glenn Burgess

**Surveillance Systems Division
Electronics and Surveillance Research Laboratory**

DSTO-TR-0838

ABSTRACT

The computational optimisation technique, genetic programming, is applied to the analytic solution of general differential equations. The approach generates a mathematical expression that is an approximate or exact solution to the particular equation under consideration. Angeline's module acquisition (MA) and Koza's automatically defined functions (ADF) are considered and the results of some modifications are presented, including a significant result regarding a generalised crossover operator.

RELEASE LIMITATION

Approved for public release

D E P A R T M E N T O F D E F E N C E



DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

Published by

*DSTO Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury South Australia 5108 Australia*

Telephone: (08) 8259 5555

Fax: (08) 8259 6567

© Commonwealth of Australia 1999

AR-011-012

February 1999

APPROVED FOR PUBLIC RELEASE

Finding Approximate Analytic Solutions to Differential Equations Using Genetic Programming

Executive Summary

Differential equations (DE's) are ubiquitous in science and engineering. In some cases solutions can be found or approximated by standard methods, however, in many cases the equations are nonlinear and either impossible to solve or have no known solution. Genetic Programming (GP) can be a useful approach in these difficult cases. This approach also has the advantage of not requiring the derivative of the objective function, which is a great advantage for problems whose objective function is not known in an analytic form.

Genetic Algorithms (GAs) are an optimisation technique inspired by Darwin's theory of evolution. The algorithm starts with a population of random parameter vectors or 'individuals' and uses these to evolve a particular individual that solves or partially solves some optimisation problem. Evolution is implemented by artificially selecting individuals from each generation based on their performance on some function that defines fitness. Each new generation is produced using the selected individuals from the previous generation and modifying their structure in various ways.

In Genetic Programming each individual is actually a program whose fitness is defined by the effects of its own execution. Genetic programming is a much more powerful technique than GAs because the encoding is more expressive and the algorithm is thus able to find solutions that were not anticipated by the designer.

This report is based on the author's Honours thesis and project. That project developed an algorithm that used Genetic Programming to find approximate symbolic solutions to arbitrary differential equations. The output of the algorithm is a population of algebraic expressions built from functions such as addition, multiplication, division and exponentiation and terminals such as variables and numeric constants. Each individual is an expression that can be evaluated numerically at a number of points in a given range so that the relative accuracy of each individual can be compared.

In principle GP can find arbitrary algebraic relations and does not find 'insoluble' differential equations any harder to approximately solve than equations with known nontrivial solutions. In addition, an approximate closed form solution is often of more value than a more accurate numerical solution for a particular instance of that problem.

The technique is applied to a number of differential equations of increasing complexity in one and two dimensions. Comparative results are given for varying several parameters of the algorithm such as the size of the calculation stack and the variety of available mathematical operators.

The results show that Genetic Programming is an effective technique that can give reasonable results, given plenty of computing resources. The technique used here can be applied to higher dimensions; although in practice the algorithmic complexity may be too high.

Several novel approaches gave negative results. One result of significant theoretical interest is that the syntax-preserving crossover used in Genetic Programming may be generalised to allow the exchange of n -argument functions without adverse effects.

Authors

Glenn Burgess

Surveillance Systems Division

Glenn studied Physics at the University of Queensland where he was awarded the Bachelor of Science in 1994. He then began his career at the DSTO in the Microwave Radar Division (now Surveillance Systems Division). He has worked on survivability and effectiveness studies for the AEW&C project and performed operational analysis for various other projects.

In 1997, he was funded by SSD to undertake Honours in Physics at Flinders University of South Australia, which he completed successfully in the same year. The thesis for that course forms the basis of this report.

Contents

CONTENTS	1
1. INTRODUCTION	1
1.1 Genetic Algorithms	1
1.2 Genetic Programming	3
1.2.1 Representation Language.....	3
1.2.2 Algebraic Expressions.....	3
1.3 The Project.....	4
2. AIM.....	5
3. MATERIALS	5
4. METHOD AND RESULTS	6
4.1 Reverse Polish Notation	7
4.1.1 RPN Calculation	7
4.1.2 Functional Interpretation	8
4.2 Allowed Functions.....	9
4.3 Language Expressiveness	10
4.4 Crossover	11
4.5 Real-valued Constants	12
4.6 Evolution.....	13
4.7 Fitness.....	14
4.7.1 Comparing Performance	15
4.8 Examples.....	15
(i) Sine(t), $0 < t < \pi/2$	15
4.8.1 Performance Measures	16
(ii) Sine(t), $0 < t < 3/2 \pi$	18
(iii) Sine(t), $0 < t < 5/2 \pi$	18
4.9 Stack Size.....	19
4.10 Function Set	20
4.11 Extending to More Dimensions	21
(i) Circular Boundary.....	21
(ii) Non-circular Boundaries	23
5. NOVEL APPROACHES	24
5.1 Syntax-Preserving Crossover.....	24
5.2 Automatic Definition of Functions	25
5.3 Module Acquisition	25
5.4 Continuous Module Acquisition.....	26
5.5 Chromosome-Matched Crossover.....	27
6. DISCUSSION	28
7. CONCLUSIONS AND FURTHER WORK.....	29
8. BIBLIOGRAPHY.....	30

APPENDIX A: PROGRAM LISTINGS.....	31
expo1.m.....	31
first.m	32
gpdepth.m	32
gpdcode.m.....	33
gpdeval.m.....	33
gpdop.m.....	34
gpeval.m	35
gpfindsb.m.....	35
gpinit.m.....	37
gpmain.m.....	38
gpmain.m.....	39
gpmut.m.....	40
gpmutate.m	41
gpop.m	41
gprelax.m	43
gprun2.m	43
gprun4.m	48
gpxexchg.m	54
gpxfunc.m.....	55
gpxinsrt.m.....	57
insort.m	58
last.m	58
meanmask.m	59
nerf.m	59
sin2d.m.....	59
sine1.m	61

1. Introduction

Genetic Programming (GP) is a general computational technique that was first suggested by John Koza in 1989 [2]. Genetic Programming is a variant of Genetic Algorithms (GAs), a biologically-inspired computational technique invented by John Holland in 1975 [1]. The fundamental ideas of both these fields come from Biology, and much of the terminology used in them reflects this fact. I will try to define this terminology as I use it.

1.1 Genetic Algorithms

Genetic Algorithms are an example of a 'soft' optimisation technique. That is, they provide a general approach requiring a minimum of domain-specific knowledge, but they are applicable in a wide range of areas. A 'hard' technique on the other hand is one which has a lot of domain-specific knowledge built-in. For any particular problem, if a hard technique is available then it will often perform better (faster, more accurate solution) than a soft technique, but it will be less flexible and has probably been designed specifically for the job. GAs are most likely to be useful in problem domains for which there are no good techniques already. The great advantage of GAs is that they can search an extremely large search space relatively efficiently. GAs have been shown to outperform a range of hillclimbing algorithms on a class of problems [3]. These problems are hard for the hillclimbers because of the extreme length of the path to the optimum. Under certain conditions GAs are actually equivalent to generalised simulated annealing [4], an optimisation technique with known, good convergence properties.

Genetic Algorithms are inspired by Charles Darwin's principle of natural selection. The basic algorithm starts with a 'population' of random parameter vectors or 'individuals' and uses these to evolve a particular individual that solves or partially solves some optimisation problem. 'Evolution' is implemented by using an artificial selection mechanism at each generation and using the selected individuals to produce the next generation. New individuals can be produced from old individuals by a variety of means including random perturbation (asexual reproduction or 'mutation') or by combining parts from two or more individuals (parents) to make a new individual (hermaphroditic reproduction or 'crossover').

The process of constructing new individuals from the previous generation is called 'reproduction', and mutation and crossover are called reproduction 'operators'. An individual's actual encoding (ie. the integers and reals, or just ones and zeros) is sometimes called its 'genetic' encoding and the form of the encoding is often described as consisting of 'chromosomes' or sometimes, 'genes'. The term 'representation' in GAs describes a higher level concept than the actual encoding of ones and zeros—the 'representation' is the meaning attached to each of the parameter values in the parameter vector.

The only domain-specific information available to the algorithm is implicitly built into the 'fitness function'. The representation only acquires the meaning we want if we build that assumption into the fitness function. The fitness function takes a single parameter vector (an 'individual') as its argument and returns (usually) a single 'fitness value', so called because it measures the 'fitness' of that individual in terms of its ability to solve the optimisation problem at hand. Selection occurs solely on the basis of these fitness values. This allows the neat partitioning of the genetic algorithm into three separate components:

1. The (initially random) parameter vectors
2. The (problem specific) fitness function
3. The evolution mechanism (consisting of selection and reproduction)

In GAs the evolution mechanism is often generalised to such an extent that the *only* change required to apply the algorithm to a new problem is to define a new fitness function. A number of crossover operators may be available, but many of these are identical in principle, but for a change in representation¹. A common choice is 1-point crossover which chooses a point randomly along the parameter vector and concatenates that part of the vector to the left of this point in one parent's encoding with that part to the right in the other parent's encoding. This can be generalised to n-point crossover by choosing n points and taking each section from alternate parents.

It is important to simulate a sufficiently large population ($\sim 10^2$ - 10^3), and to allow evolution to proceed for enough generations (~ 50 - 500), however, the success of a genetic algorithm depends mainly on how appropriate the reproduction operators are for the particular representation, and how informative the fitness function is. An 'appropriate' reproduction operator is one which produces new individuals in such a way that there is a reasonable probability that the fitness of the new individual will be significantly higher than that of its parent(s). An 'informative' fitness function is one that gives monotonically increasing fitness values to individuals as they get closer to solving the problem under consideration. The combination of these two properties opens the way to the successful evolution of a solution to the optimisation problem at hand. Much of the theoretical work in the field of GAs is into finding the most appropriate reproduction operators and representations.

The range of behaviours generable by a GA (or GP) is limited explicitly by the fitness function and the meaning of the parameters it accepts. In GAs the meaning of the parameters is usually quite limited and the number of parameters fixed. In Genetic Programming, the representation of each individual is actually a *program* in some limited programming language which must be *executed* to determine the fitness of the individual². In this case the fitness function contains an interpreter that must decode

¹ Indeed, under certain conditions the effects of n-point crossover can be made identical to those of n-point mutation in another, carefully designed representation [8].

² The difference between GA and GP noted here is the main difference in practice, however, it is largely traditional. The only *conceptual* differences between the two are the normally fixed chromosome length of GAs and the syntax-preserving crossover operator usually used in GP [5].

the instructions of each individual to determine how well that individual solves the given problem.

1.2 Genetic Programming

As in GAs, the encoding of each individual in GP is a string of numbers. However, each program is represented in a tree structure (Figure 1) corresponding to the syntax of the operations it uses. It is important that the random modifications of the reproduction operators do not make programs that are not syntactic. If they did then many reproductions would result in 'dead' offspring. To prevent this, GP uses a syntax-preserving crossover operator. This works by taking an entire subtree from one parent and replacing it with an entire subtree from the other parent. The subtrees are selected randomly, as for 1-point crossover in GAs, but now there is no need to use the same point from each parent because the new individual need not have the same length as its parents.

Genetic programming is a much more powerful technique than GAs in the sense that the encoding is much more expressive and it is much more likely to find solutions that were not anticipated by the designer. Genetic Programming is normally regarded as being a specialisation of GAs. I will thus use the term GP when discussing facts that apply specifically to that field, and the term GA when describing more general properties shared by both, unless otherwise specified.

1.2.1 Representation Language

The choice of the programming language that the representation uses (the 'representation language') is completely arbitrary and in many cases is actually constructed specifically (since the interpreter must usually be written by hand). The number and kind of programming instructions available defines the expressive power of the language (in terms of the Chomsky hierarchy), but the range of behaviours evolvable may be less than this. The actual behaviours evolvable in practice is intimately tied to the structure of the representation language and the actions of the reproduction operators. Some constructions may never evolve in practice because they are just too difficult to find or they do not offer sufficient rewards along the way.

1.2.2 Algebraic Expressions

A simple example of the kind of representation language that might be used is the domain of algebraic expressions. For instance, we might want to design an algorithm that can discover trigonometric identities, or an approximate expression for π , or find a symbolic expression that fits a set of data (symbolic regression). For these problems we need to represent mathematical functions as algebraic formulae containing variables and numerical constants. In the terminology coined by Koza [6] the operations are referred to as functions and the variables and constants are 'terminals'.

So for the symbolic regression example given above we might allow functions such as addition, subtraction, multiplication, etc. and terminals such as variables denoting the data to be fitted, and a number of random real constants. The 'programs' evolved can be interpreted as functions acting on the data. The fitness of each individual can be determined by evaluating the function represented by its encoding, for every ordinate of the data set, and comparing the calculated value with the data.

For instance, suppose the data to be fitted are,

$$y=[0.0,0.17,0.32,0.49,0.65,0.76,0.88,0.95,1.00], \text{ against}$$

$$x=[0,10,20,30,40,50,60,70,80],$$

and the approximated equation is:

$$y' = 1 - 0.00024 \times ((80-x)^2 - 10 \times (80-x))$$

Thus $y' = [-0.3440, -0.0080, 0.2800, 0.5200, 0.7120, 0.8560, 0.9520, 1.0000, 1.0000]$, evaluated on x .

Then we find the RMS deviation of y from y' is 0.4162.

This value can be regarded as the raw fitness value associated with this problem and can be used as the basis for selection between two or more competing individuals.

The technique for symbolic regression can be easily generalised to apply to the solution of any equation, provided suitable notice is taken of boundary conditions. Suitable choices of fitness function could turn the problem into one of:

- symbolic differentiation: $f = \sqrt{\langle (dy/dx - y')^2 \rangle}$, where dy/dx is estimated by numeric differentiation
- symbolic integration: $f = \sqrt{\langle (\int y \cdot dx - y')^2 \rangle}$, where $\int y \cdot dx$ is estimated by numeric integration
- solution of arbitrary equations: $f = \sqrt{\langle (\text{LHS}(y') - \text{RHS}(y'))^2 \rangle}$

1.3 The Project

In particular, the present work is concerned with finding approximate symbolic solutions to arbitrary differential equations. The topic of solving differential equations with GP was introduced by Koza in [6]. It is really just a particular instance of Genetic Programming. Koza demonstrates for example the successful evolution of the correct solution, $e^{-\sin x}$, to the differential equation,

$$dy/dx + y \cos x = 0; \quad \text{with boundary condition, } y(x=0) = 1.$$

This is achieved after 6 generations of one particular run with a population of 500 individuals. It should be stressed that this is an unusual occurrence. More usually a correct or good solution will be found only within 20-40 generations. Frequently the 'correct' solution is disguised in a form that is only identical after transformations (eg.

expansions and factorisation), and sometimes no good solutions will be found in a run at all (ie. the algorithm must be reseeded and run again).

Differential equations are ubiquitous in science and engineering. In some cases solutions can be found or approximated by standard methods, however, in many cases the equations are nonlinear and either impossible to solve or have no known solution. GP is a potentially useful approach in these difficult cases. In principle GP can find arbitrary algebraic relations and should not find 'insoluble' differential equations significantly harder to approximately solve than equations with known nontrivial solutions. This approach has the advantage of not requiring the derivative of the objective function, which is a great advantage for problems whose objective function is not known in an analytic form. In addition, an approximate closed form solution may be of more value in many circumstances than a more accurate numerical solution for a particular instance of that problem.

There are many examples where GP techniques may be useful. For instance, finding stable toroidal plasma equilibria of magnetic field configurations, or the inverse problem: finding magnetic field configurations with stable toroidal plasma equilibria.

2. Aim

The aim of this project is to implement an algorithm using Genetic Programming that can produce approximate symbolic solutions to arbitrary differential equations. The candidate solutions should be mathematical expressions that can be evaluated numerically at a number of points in a given range so that the relative accuracy of several such solutions can be compared.

The scope of the project is to:

- produce an algorithm with comparable performance to published work on one-dimensional problems
- apply the algorithm to a non-trivial two-dimensional problem
- consider and experiment with variations to GP algorithm

3. Materials

In principle the particular hardware and software used is irrelevant, however, these factors have bearing on issues of wall-clock efficiency and scalability.

The present system is implemented in Matlab™. Matlab is a high-level interpreted language that makes use of efficient compiled library subroutines to perform common vector and matrix operations.

The program is written entirely in Matlab V4.2 compliant code. It runs without modification on PC or Silicon Graphics (SGI) platforms. All of the timed measurements

were performed on a SGI Origin 2000 consisting of 2xR10000 processors. Matlab is an interpreted language but it includes the facility to call external routines from within it. A Matlab to C 'compiler' (mcc) for V4.2 was used to produce compiled versions of several key functions, to give a relatively cheap speed-up. Matlab V5 is a much more elegant language but it was not used because, at the time this work was done mcc was not yet available for V5, and without this compiler the program runs significantly slower ($\sim \div 2$).

4. Method and Results

I have used 'generational' Genetic Programming. In this paradigm a population of new individuals is produced from and replaces the old population all at once. In contrast, in 'steady state' GP, individuals are selected for mating and for replacement one at a time or in small groups. This causes the population to vary much more smoothly, but is slower to evolve overall (particularly in Matlab).

GAs are usually implemented using a form of selection known as *proportionate selection*. This means that the probability of any individual propagating into the next generation is proportional to its fitness. Put another way, the fittest individuals produce the most offspring and the least fit individuals die out. Due to the computational requirements of simulating even simple problem domains we are usually restricted to a few hundreds of individuals in the population at one time. The effect of proportionate selection on a population of this size is that the population quickly loses 'genetic diversity', that is, the most successful individual quickly dominates the population and all individuals eventually come to closely resemble each other. Loss of diversity is a serious problem for GAs because it severely reduces the volume of the search space spanned by the population at any one time. This significantly decreases the overall efficiency of the algorithm and may prevent the algorithm finding a good solution at all.

My implementation was based on a scheme devised by Culberson in 1994 [8] called Gene Invariant Genetic Algorithm (GIGA). In this experimental system the reproduction operators and selection scheme are defined in such a way that genetic diversity in the population remains constant. Each generation is ranked according to fitness and then paired off in descending order so that each individual mates with an individual of similar fitness. Genetic diversity is maintained because every crossover that occurs creates two offspring from each pair of parents, each containing corresponding parts from opposite parents. For 1-point crossover this means cutting both of the parents into two parts and exchanging the parts to get two new individuals with none of the genetic material lost. For the syntax-preserving crossover operators used in GP it means the parents exchange one complete subtree.

4.1 Reverse Polish Notation

GP is often implemented in LISP, a functional programming language that uses nested lists as the basic data structure so it is very easy to extract and exchange subtrees. LISP uses prefix notation for specifying functions. I have used a non-standard representation for my implementation: each program is encoded in Reverse Polish Notation (RPN), a common standard for engineering calculators. RPN is similar to the post-fix notation used sometimes in computer science, but RPN uses no parentheses. Each function symbol applies to the previous two terminals or subtrees in the expression. RPN allows the evaluation of any ordinary arithmetic expression without the use of parentheses.

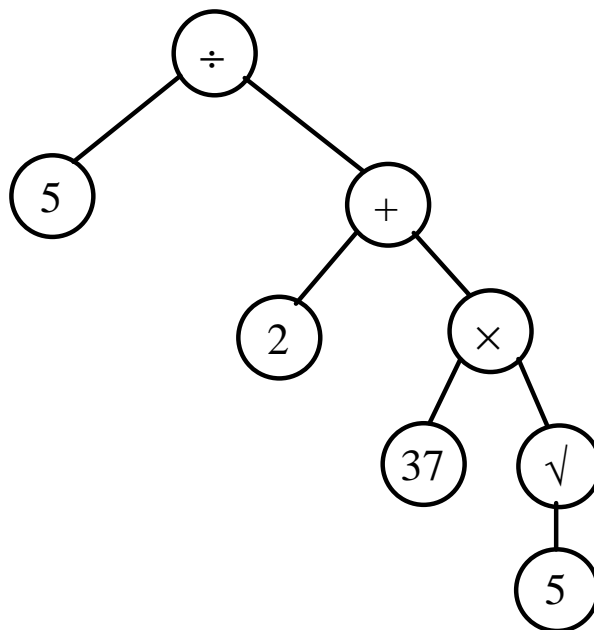


Figure 1. Syntax tree for expression $5 \div (2 + 37 \times \sqrt{5})$, (in-fix); or '5,2,37,5, $\sqrt{\quad}$, \times , +, \div ', (RPN).

For example, the expression, $5 \div (2 + 37 \times \sqrt{5})$, is shown in tree form in Figure 1, and it can be written in RPN as, '5,2,37,5, $\sqrt{\quad}$, \times , +, \div '.

4.1.1 RPN Calculation

An RPN calculator uses a small stack (~ 4 numbers) to keep track of the partial results. Table 1 traces through the contents of the stack as a simple expression such as that shown in Figure 1 is evaluated.

Operation	Stack Contents, x	y	Z	w
5	5	—	—	—
2	2	5	—	—
37	37	2	5	—
5	5	37	2	5
$\sqrt{}$	2.236	37	2	5
\times	82.735	2	5	5
+	84.735	5	5	5
\div	0.059	5	5	5

Table 1. Evaluation of RPN expression, '5,2,37,5, $\sqrt{}$, \times ,+, \div '.

The stack is really just a list of four numbers that are shifted down one place when a new terminal is input, or up one if a two-argument function symbol is input. A one-argument function such as square-root only affects the most recent number and thus does not move the stack. Initially the stack is not empty, but contains 'random' (ie. unspecified) values. If more numbers are put into the stack than it can hold, then the earlier numbers simply fall off the bottom. When the stack moves up, the bottom element is effectively copied, so the stack never 'runs out' of numbers, it just keeps giving whatever numbers its registers contain. In this way, all RPN expressions can be made syntactic. They may not be well defined, such as when 'random' numbers come out of an empty stack, but that doesn't stop them giving a result.

4.1.2 Functional Interpretation

There is a sense in which any RPN expression evaluated in this manner may be interpreted as a mathematical function acting on the contents of the stack, and placing its output(s) on the top of the stack. For instance, in the example above, the RPN subsequence, '5, $\sqrt{}$, \times ', can be regarded as a function,

$$f: \mathfrak{R} \rightarrow \mathfrak{R}, f(x) = (x \times \sqrt{5}),$$

where x is the number contained in the top of the stack before the evaluation (37 in this case), and

$f(x)$ is the value left on the top of the stack afterwards.

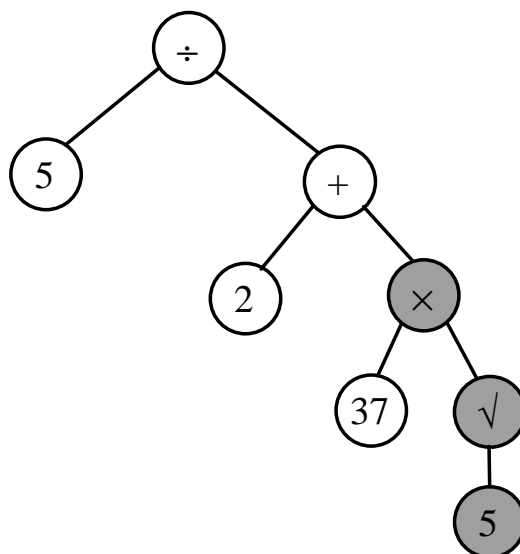


Figure 2. Syntax tree for expression '5,2,37,5,√,×,+,÷', (RPN); interpretation of '5,√,×' as a function.

Figure 2 depicts graphically the relationship between a 1-argument function and its argument, for the same expression as in figure 1. The shaded nodes are those nodes constituting the body of the function.

In this manner each individual may be regarded as a function which takes some number of arguments and returns some number of arguments, by passing and retrieving values in and from the stack. Parameters are passed to the RPN programs by placing the values of the parameters in the top elements of the stack before the program is run, and the output is defined to be the number on the top of the stack when the program terminate. These definitions constitute part of the 'representation' that must be discovered by the GP individuals. ie. they must discover that the numbers already in the stack are relevant to the solution of the problem, and that the answer must be left in the stack after execution.

4.2 Allowed Functions

Matlab is designed for efficient manipulation and calculation with matrices and vectors. This allows each program defined in RPN as described above to be evaluated relatively efficiently on an entire set of parameter values. The allowed functions can be specified individually for each experiment. The range of choice includes (but is not limited to) addition, subtraction, multiplication, division, exponentiation, natural logarithm, sine, cosine, unary minus (+/-), unary division (1/x), as well as a few 'kludge' functions that are often available on RPN calculators such as, rotate stack up/down one register, store/retrieve external variable (one only).

Some of these functions are undefined for particular values of their arguments. Trying to evaluate such a function would normally constitute a 'fatal' error, on the behalf of the individual, since it is impossible to assign a fitness. To avoid this problem (which simply reduces overall efficiency) Koza [6] defines a number of 'protected' functions being versions of the ordinary functions modified so that they are defined for all input values. eg. Koza's protected-division function returns a value 1 if the denominator is zero, and y/x otherwise. I have sometimes used (my own versions of) protected functions, and sometimes not—Matlab is relatively forgiving of division by zero and other undefined operations.

4.3 Language Expressiveness

It is worth mentioning some formal language theory at this point. The basic RPN calculation algorithm that I have described is equivalent to a Push-Down Automaton (PDA) in the Chomsky language hierarchy (provided the stack never overflows), which is less powerful in terms of computability than a Turing Machine (TM). With the addition of external variables the RPN calculation algorithm becomes equivalent to the level of a Turing Machine (again, provided there are sufficiently many variables for the particular program) which means it can compute any function that can be computed by any other 'ordinary' computer.

In the RPN evaluation framework described above there is currently only one external variable, but the stack size may be set arbitrarily large. Thus in terms of the Chomsky hierarchy, the computational power of the programs that may be expressed by this system may be regarded as lying somewhere between that of a push-down automaton and a Turing machine. However, this can only be regarded as placing an upper bound on the computing power of programs that may be evolved in this framework using GP. For instance, it might be the case that the GP algorithm never 'learns' to use the external variable at all, or perhaps never in a computation that could not have been performed using the stack alone.

4.4 Crossover

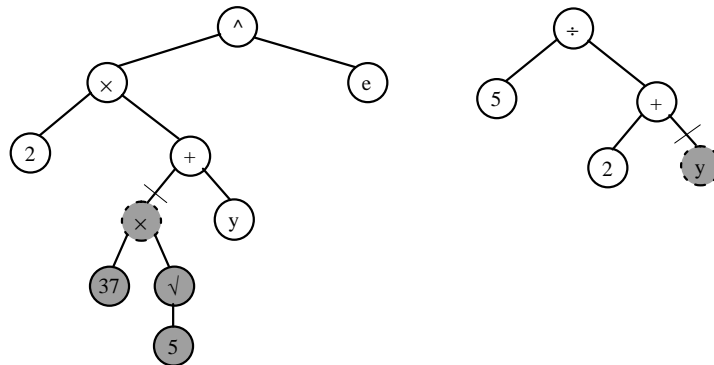


Figure 3. Syntax trees for $(2 \times (37 \times \sqrt{5} + y))^e$, $[2,37,5,\sqrt{},\times,y,+, \times,e,^]$ and $5 \div (2+y)$, $[5,2,y,+, \div]$.

The action of the crossover operator can now be shown explicitly. Figure 3 shows the expressions $(2 \times (37 \times \sqrt{5} + y))^e$ and $5 \div (2+y)$ in tree form. Suppose these two expressions are selected to undergo crossover. In each expression a node is randomly selected for the crossover (indicated with a dotted outline). The entire subtrees below and including these points (shaded) will be exchanged between the two individuals resulting in two new individuals (Figure 4). It can be seen that this operation is the same as replacing y by $37 \times \sqrt{5}$ in infix notation, or replacing y by $'37,5,\sqrt{}'$ in RPN. Thus the crossover operator can be implemented directly on the RPN expressions by exchanging the appropriate subsequences (Appendix A: gpxexchg.m). All that is required to do this is to determine the node in the tree that corresponds to the start of the required subsequence in the RPN expression. This is a relatively straightforward matter (Appendix A: gpfindsb.m).

The initial length of each individual in the population is set to a relatively low constant value. The initialisation procedure therefore ensures that each allowed function symbol occurs at least once in the initial population, and thereafter selects functions uniformly randomly (Appendix A: gpinit.m). The fixed initial length may be

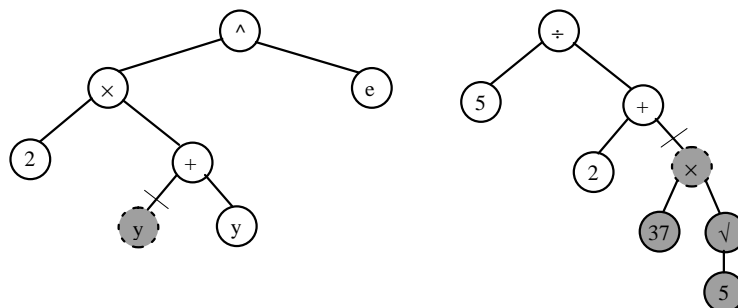


Figure 4. Syntax trees after crossover: $(2 \times (y + y))^e$, $[2,y,y,+, \times,e,^]$ and $5 \div (2 + 37 \times \sqrt{5})$, $[5,2,37,5,\sqrt{}, \times,+, \div]$.

considerably smaller than that of the optimum program so a significant increase in length must occur in order to be able to reach the optimum. In the pure implementation of GIGA the total number of nodes in the population is constant thus making it impossible to increase the average length of the population. For this reason a second crossover operator is defined that ‘inserts’ a subtree from each individual into a random place in the other one, thus increasing the average length of the population without reducing the diversity (Appendix A, gpxinsrt.m)³.

4.5 Real-valued Constants

The problem of how to include real-valued constants in GP and optimise them has been tackled in a number of different ways by different authors. Koza [6] introduces a limited number of real-valued constants at the start and allows the algorithm to find transformations and combinations of these to reach optimum values. This approach is attractive because it is *homogeneous*, ie. it allows optimisation of real constants in the same framework as the rest of the problem, however, this is an inefficient approach both in terms of computation and representation (Marenbach et al. observe of Koza’s approach in [6], “Even for simple problems, solutions filled complete pages”).

Marenbach et al. solve the problem by including variable tokens in their GP expressions [9]. The optimum variable values for each expression are found separately using a conventional numerical optimisation approach. This is a more efficient approach since it enables the task of optimising the parameters to be handed over to a known efficient technique especially designed for this task.

My own approach is partway between these two approaches. For each individual I define a vector of ‘post-multiplier’ constants corresponding to the RPN expression of the individual. These constants are used during the RPN evaluation, and as the name suggests, the result of each RPN instruction is multiplied by the corresponding constant. Thus if the expression, $5 \div (2+y)$, $[5,2,y,+, \div]$, has postmultiplier-constant vector, $[-0.013, 1.02, 4.56, -2.10, 0.64]$, then it is in fact evaluated as, $[5, -0.013, *, 2, 1.02, *, y, 4.56, *, +, -2.10, *, \div, 0.64, *]$ in RPN, or $(5 * (-0.013) \div (2 * 1.02 + y * 4.56)) * (-2.10) * 0.64$ in infix notation.

³ Strictly speaking, the action of this ‘insertion’ operator is more involved than merely inserting subtrees. The operation inserts a RPN subsequence defined in the same way as for the exchange operator, but the effect on the parent tree shifts all the nodes attaching to nodes above the point of insertion. In terms of RPN evaluation the operation pushes a new value onto the stack in the middle of a half-completed calculation. In summary the effect is quite unpredictable, however, like all the other operators in this implementation, insertion is only used if it leads to an increase in the fitness of at least one of the two offspring (see later).

This leads to an interesting diversion regarding GAs in general. Because so much of the algorithm is arbitrary or stochastic such as the encoding and the reproduction operators, many program errors can go unnoticed for a long time. This awkward situation has the unusual side-effect of making the performance of the algorithm as a whole more robust to programmer error.

Each program instruction in an expression is associated with a unique constant that is transferred and replicated by the actions of crossover so that each constant can be regarded as having a fixed or nearly fixed interpretation with regard to the value returned by the expression. That is, the constant 4.56 is always applied to the value of y , and 0.64 is always applied to the result of the division.

The postmultiplier constants are themselves modified by a distinct asexual reproduction operator that applies normally-distributed random perturbations to the values of the constants (Appendix A, `gpmutate.m`). The magnitude of the perturbations should be *variable*, because we hope that the values of the constants will converge to some optimum values, and *customised* because some constants may converge faster than others, or be near zero. We thus define another vector of real parameters (called the reproduction parameters, collectively), where these values have the literal meaning of the magnitude of Gaussian noise added to each postmultiplier constant when the constant-adaptation (mutation) operator is used, and the implied meaning that they are proportional to the approximate error in the corresponding postmultiplier constant.

This arrangement can be considered to allow a kind of random-walk hill-climbing algorithm to optimise the values of the postmultiplier constants. The concept of including the parameters to the operator in the representation (as is effectively the case with the reproduction parameters) is an example of what is called co-evolution of operator settings in GAs, but is referred to as Self-Adaptation in the Evolutionary Strategies community (ES—another offshoot from GAs). Such an approach has been reported to give faster convergence in several experiments [10], however, other experiments have reported a decrease in performance [12].

4.6 Evolution

The Gene Invariant Genetic Algorithm (GIGA) arranges all individuals in order of decreasing fitness and then mates each adjacent pair (ie. every individual mates once per generation). The offspring from each mating contain the same genetic material as their parents, just cut and arranged differently. After mating the offspring are ranked again and the process is repeated. In general, the average fitness of the population will neither increase nor decrease. The *range* of fitnesses in the population should increase, however, so that eventually the top individual solves the problem.

GIGA is an experimental algorithm that demonstrates that GAs can work without selection. However, it is computationally expensive to retain a population of individuals that may in practice have nothing to contribute to the ‘correct’ solution. Thus my implementation does use selection—a kind of selection known as *soft-brood selection*. In this paradigm, selection occurs on the set of offspring from every mating rather than on all offspring taken together. This presents a ‘softer’ selection criterion that ensures that some offspring survive from every pairing or ‘brood’ thus retaining diversity for longer.

Soft-brood selection is implemented, not by producing several offspring with the same operator, but by producing a pair of offspring using each available operator (there are four, counting the 'clone' or no-change operator). In each generation the individuals are paired off as for GIGA, but with slight 'mixing', ie. the individuals are ordered and then the ordering is perturbed by the addition of a normally distributed random number to the index and reordering in the new indices. This process is found to increase the efficiency of the algorithm significantly by bringing different individuals into contact. A common situation otherwise is for the population to become sorted into strata corresponding to identical classes of solution. A typical perturbation amplitude is 10.

In each generation then, individuals are paired and mated according to the perturbed ordering. Each operator is applied to each pair resulting in 4 pairs of individuals vying for selection for the next generation. The winning pair is the pair containing the fittest pair. Thus if none of the pairs of offspring contain a single individual that is fitter than both of the parents then the 'clone' operator is used⁴, ie. the parents return to the population.

4.7 Fitness

The 'raw' fitness measure I use is determined primarily from the mean square of the difference between the numerically estimated values of the RHS and LHS of the differential equation, evaluated at a set of points in the valid range. The normal way of including boundary conditions in the formulation is to add extra terms into the fitness calculation that have a minimum value when the boundary conditions are satisfied. In practice each of the terms in the fitness calculation must be weighted by a factor proportional to the relative importance of that factor to the others. These weights are generally determined by trial and error.

An alternative to this approach is to 'normalise' or post-process the result of each individual to *ensure* that the boundary conditions are satisfied. eg. for $d^2y/dx^2 = -y$, $y(0)=0$; $y(\pi/2) = 1$, procedure is to subtract $y'' = y' - y'(0)$, then divide by $y''(\pi/2)$. This has the advantage of making most individuals instantly reasonable approximators to the desired solution. It has the obvious disadvantage that the expressions thus generated do not, of themselves, solve the problem. The meta information of the normalisation procedure is required to convert the actual expressions into the approximate solutions. Also, the normalisation procedure can have discontinuous effects near the origin, or in other regions, particularly if not defined carefully. The

⁴ For some problems the clone operator is used very frequently, particularly in the second half of the run. This constitutes wasted computation if the generated offspring are ever the same as those seen before, which is possible if the individuals are not being replaced. To avoid this, then, a small bias is added to the fitness of the parents to bias against choosing the no-change operator. This may also help prevent getting trapped in local optima.

approach I have used is to apply a normalisation as described, and also include boundary condition terms in the fitness function, but with low weights.

Other contributions to the fitness include a term to penalise the ‘trivial’ solution (ie. all zero, can be important for some problems), and a term related to the entropy (for minimisation of the description length) with a low weight.

4.7.1 Comparing Performance

The stochastic nature of the GP technique means that to determine whether any particular change to the algorithm results in an improvement it is necessary to perform each run a number of times (~ 20). Also, some changes such as increasing the population size, may result in improved performance per *generation*, but only by taking a longer time to evaluate per generation. In order to factor effects like these out we need to be able to compare algorithms’ accuracy of solution on some common fitness scale, after having used the same total amount of computing power. CPU time is an excellent measure of computing resources used for the present purposes since that variable is unaffected by any other jobs running at the same time. Thus note, for the plots that follow comparisons of algorithmic efficiency can only be made for identical problems after using the same amount of CPU time, or upon reaching the same average fitness.

4.8 Examples

(i) Sine(t), $0 < t < \pi/2$

The first problem that was attempted was the differential equation, $d^2y/dx^2 = -y$, $y(0)=0$; $y(\pi/2) = 1$ (Annex A, gprun2.m). Figure 5 shows the (normalised) top 50% of the population after a single run (population 150, 100 generations) for this problem. The solid line shows the actual solution ($\sin x$), the dashed line is the best solution found and the dotted lines are the remaining individuals in the top 50%. Notice that the normalisation procedure constrains all the lines to satisfy the boundary conditions.

The mathematical expression for the best individual in this run is equivalent to:

$$\begin{aligned} & (1.389) \cdot (0.01738) \cdot (-1.38) \cdot (0.1775) \cdot ((-1.841) \cdot (0.02282) \cdot (0.7963) \cdot 1 \cdot t1 - \\ & (-1.102) \cdot (-1.178) \cdot (-0.07694) \cdot (-0.9438) \cdot (1.948) \cdot (0.1781) \cdot \\ & (-1.841) \cdot (0.02282) \cdot (0.7963) \cdot 1 \cdot t1 / ((-1.24) \cdot (-0.9438) \cdot (1.948) \cdot (0.1781) \cdot \\ & (-1.841) \cdot (0.02282) \cdot (0.7963) \cdot 1 \cdot t1.^2 + e) / ((-1.311) \cdot 1) / ((-1.462) \cdot 1)), \\ & \text{(unnormalised — to normalise must subtract } y(0), \div y(\pi/2) \text{)}, \end{aligned}$$

where $t1$ is the range variable, and $e = 0.00001$ is a small value (implements modified division).

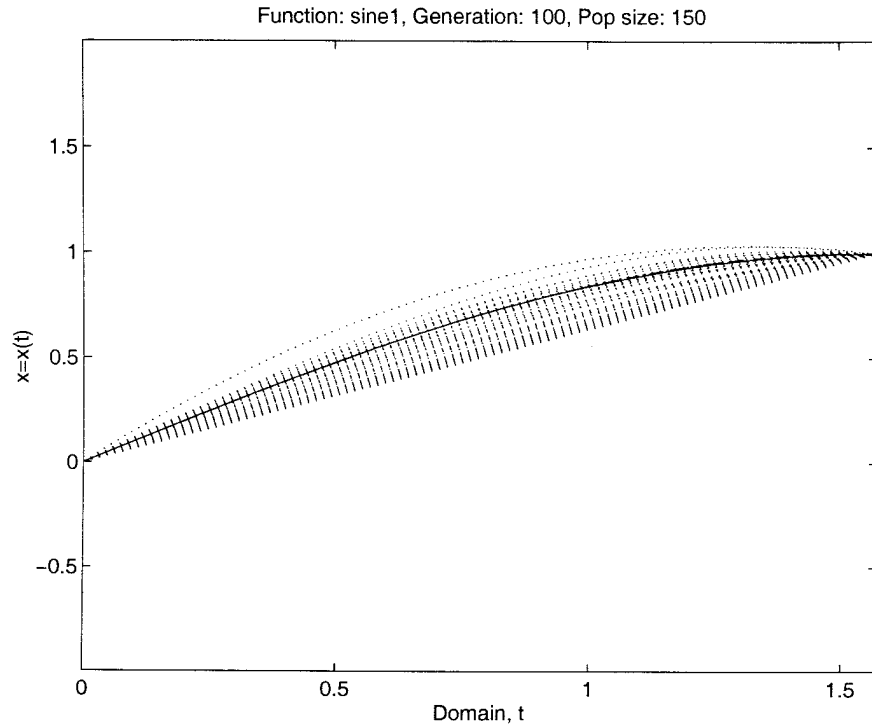


Figure 5. Top 50% of population. Problem: $y''=-y, (0,\pi/2)$.

4.8.1 Performance Measures

A number of other statistics are gathered during the learning process to help determine how efficiently the algorithm is working. Figure 6 shows the proportional use of each reproduction operator over 100 generations (ie. which operator tended to produce the highest fitness single offspring). In this case mutation (constant-adaptation) was the dominant operator for most of the run. In most cases either exchange or mutate is dominant.

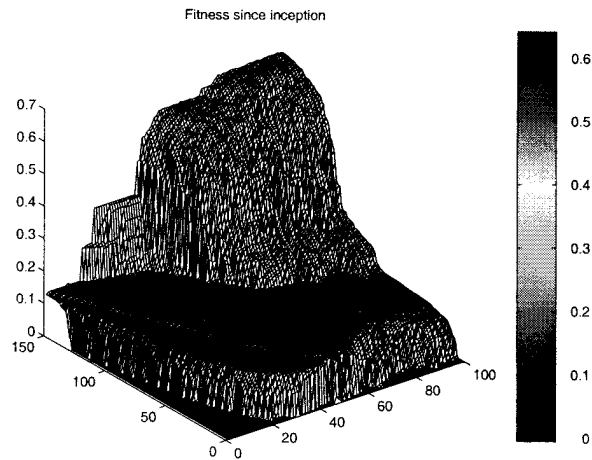


Figure 6. Fitness map. Problem: $y''=-y, (0,\pi/2)$.

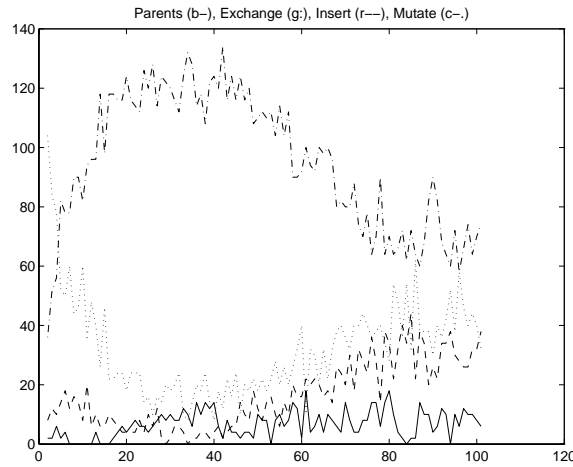


Figure 7. Reproduction operator statistics.
Problem: $y''=-y, (0,\pi/2)$.

Another useful statistic shows (in arbitrary fitness units—higher is better) the evolution of the fitness profile of the entire population (Figure 7). Here the left axis is the ordered population, the right axis is the number of generations. This graph is useful because plateaus show where a class of identical (or nearly identical) individuals coexists. An efficient algorithm has a gradually increasing slope in both positive axis directions.

We ran this experiment 20 times and collated the mean fitness (measured by standard deviation from the **actual** solution) as a function of CPU time. The results are shown in Figure 8. The solid line is the mean, the dashed lines denote the error in the mean and the dotted lines are the maximum and minimum values. This shows strong convergence up to about 60 sCPU.

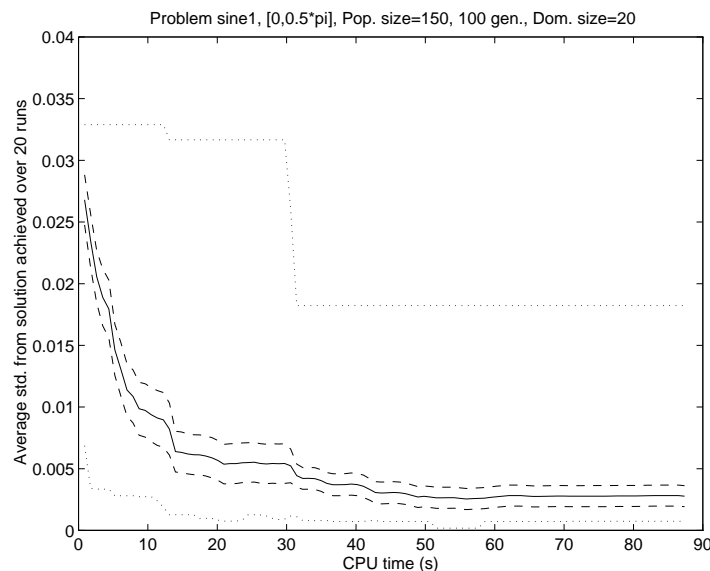


Figure 8. 'Learning Curve' for $y''=-y, (0,\pi/2)$. Solid line is mean, dashed lines error in mean, dotted lines max and min.

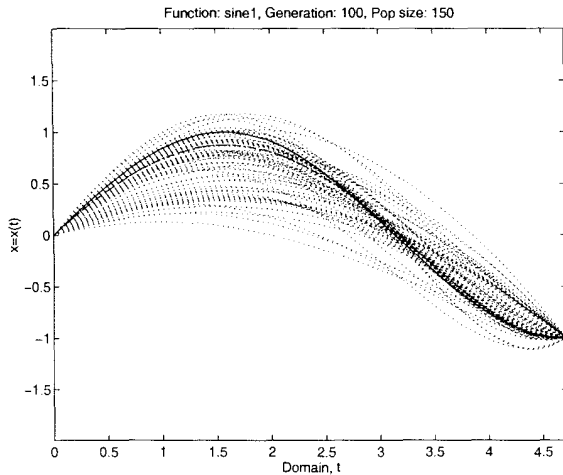


Figure 9. Top 50% of population. Problem: $y'' = -y, (0, 3\pi/2)$.

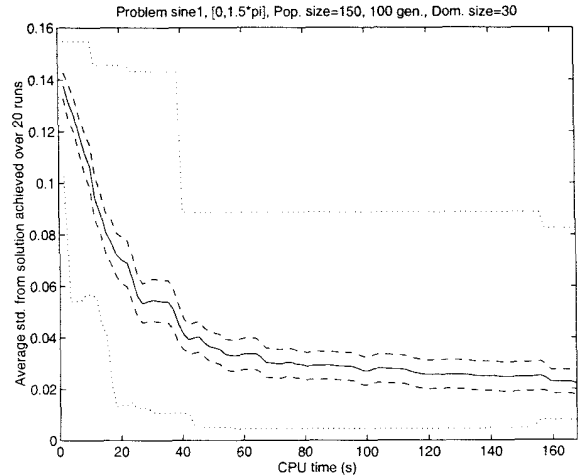


Figure 10. 'Learning curve' for $y'' = -y, (0, 3\pi/2)$, showing mean, error in mean, max and min.

(ii) Sine(t), $0 < t < 3/2 \pi$

Figure 9 shows the top 50% of the population for a sample run for the same problem on the larger range ($0, 3\pi/2$). The figure reveals a good visual agreement and a good range of curves in the top 50% of the population. In this case the best program corresponds to the following Matlab expression:

```
(0.2396) .* (0.7699) .* (-0.8016) .* (0 - (-0.8095)) .* (0 - (-0.4073)) .* ((-0.8168) .*
(1 - (0.9229)) .* (-1.077)) .* (t1 - (1.589)) .* 1) .* (0.2491) .* (-1.077) .* (t1 - (1.589)) .* 1) -
(0.5584) .* (0.2741) .* (-0.8168) .* (1 - (0.9229)) .* (-1.077) .* (t1 - (1.589)) .* 1) .*
(0.2491) .* (-1.077) .* (t1 - (1.589)) .* 1) .* (0.1848) .* (0.2741) .* (-0.8168) .*
(1 - (0.9229)) .* (-1.077) .* (t1 - (1.589)) .* 1) .* (0.2491) .* (-1.077) .* (t1 - (1.589)) .* 1) )) .*
(0.2799) .* (-0.8016) .* (0 - (-0.8095)) .* (0 - (-0.4073)) .* ((-0.8168) .* (1 - (0.9229)) .*
(-1.077) .* (t1 - (1.589)) .* 1) .* (0.2491) .* (-1.077) .* (t1 - (1.589)) .* 1) - (0.5584) .*
(0.2741) .* (-0.8168) .* (1 - (0.9229)) .* (-1.077) .* (t1 - (1.589)) .* 1) .* (0.2491) .*
(-1.077) .* (t1 - (1.589)) .* 1) .* (0.1848) .* (0.2741) .* (-0.8168) .* (1 - (0.9229)) .*
(-1.077) .* (t1 - (1.589)) .* 1) .* (0.2491) .* (-1.077) .* (t1 - (1.589)) .* 1) )) (unnormalised)
```

Figure 10 shows the corresponding learning curve for this problem.

(iii) Sine(t), $0 < t < 5/2 \pi$

A final interesting case was studied for the sine problem in the range ($0, 5\pi/2$). Figure 11 shows the top 50% of the population after a single run (population 150, 300 generations). The best individual after 300 generations in this case is the dark line that begins on the x-axis, and then rises monotonically to pass through the second boundary condition. This demonstrates a fundamental principle of GAs. The individuals will use any trick possible to satisfy the fitness function. Or, in other words, the individuals evolved are only as good as the fitness function. In this case, accuracy of solution is measured by the average square sum of the second derivative and the value of the function. Here the 'best' individual can be seen to be 'cheating', by

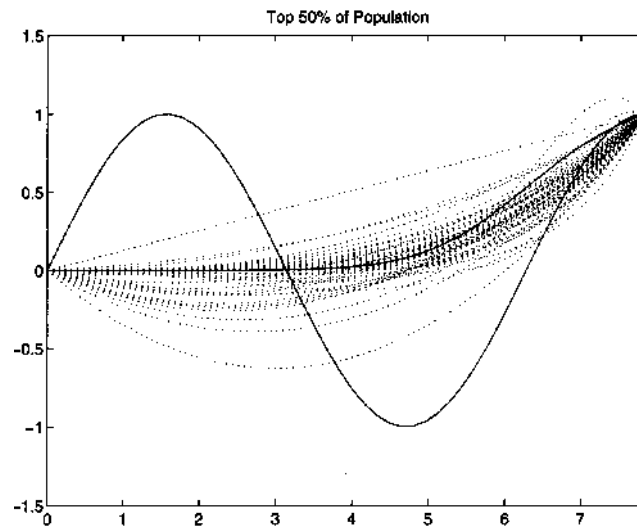


Figure 11. Top 50% of population. Problem: $y''=-y$, $(0, 5\pi/2)$.

finding the trivial solution ($y=0$) for part of the range, and then trying to satisfy the second boundary condition. The fact that the second half of this function clearly does not satisfy the differential equation is outweighed by the fact that it satisfies it very well for the first part of the range, and there are no better individuals around to challenge it.

4.9 Stack Size

The size of the stack used in the RPN calculation procedure limits the complexity of the computations that may be performed by a single individual. In the experiments above a fixed stack of four rows was used. The example in (ii) above was repeated using a stack size of six. The resultant learning curve is shown in figure 12 (compare to

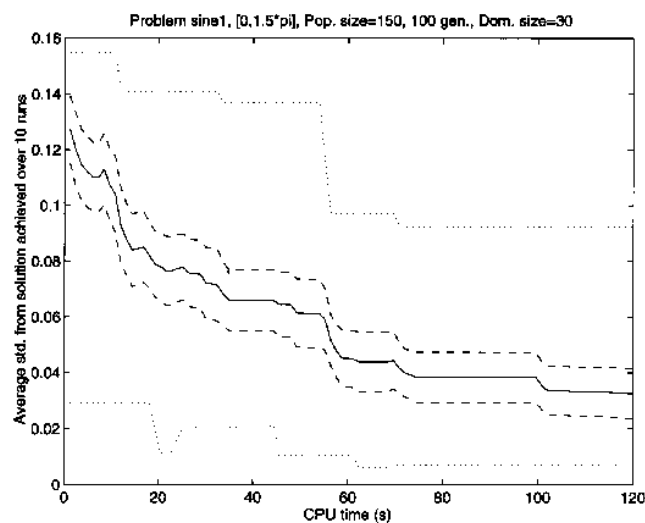


Figure 12. Learning curve with stack size = 6.

figure 10). If the values of the mean in these two curves are compared at a common CPU time such as 120 s it can be seen that the results are not significantly different. Note: this learning curve is derived from ten runs only, hence it has a greater error in the mean.

4.10 Function Set

These experiments were conducted using the functions (+, -, ÷, ×, <modified division>⁵), and the terminals 1 and x (copy top of stack). Several extensions of this basic set of functions were tested, with no significant differences observed. For example, problem (ii) was repeated using the same functions and terminals plus the following functions/terminals: <rotate stack up/down one row>, <push/pop variable>, and <push zero>. The resulting learning curve (Figure 13) shows no significant difference in performance.

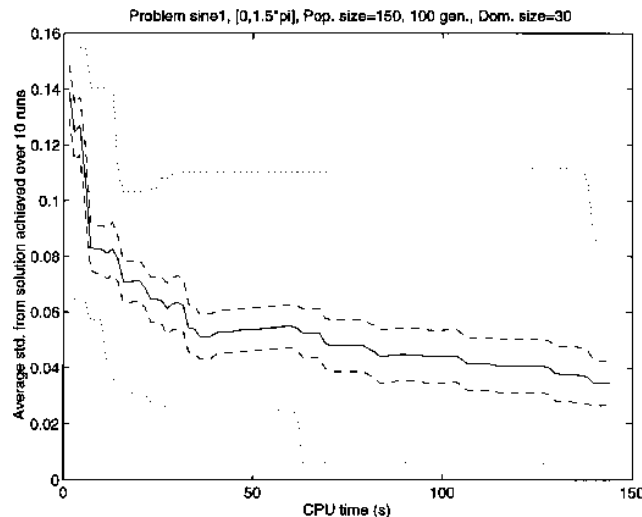


Figure 13. Learning curve with function set: [+ , - , / , * , 1 , 0 , x , V , x→V , R↑ , R↓].

On the other hand, when <sin> is included in the set of allowed functions the system very quickly converges on the correct solution. For instance, in one run the system found the expression $0.94 * \sin(t)$ after the second generation, and subsequent generations gradually refined the value of the numeric constant.

The fact that efficiency is not improved by expanding the function set suggests that the system is not well placed to use more expressive computations. Or else the increase in efficiency is cancelled by a corresponding decrease in efficiency due to the increased number of available functions to choose between, and thus greater search-space volume.

⁵ Modified division avoids division-by-zero like protected division (used by Koza). It is defined as, $x \div (y + e)$, where $e=0.00001$ is a small number.

4.11 Extending to More Dimensions

Several minor modifications were required to extend the potential of the system to generate functions of two dimensions. A two-dimensional fitness function was constructed using a representation of a mesh in polar coordinates. The function evaluates the given data on a polar grid as a solution to the equation, $\nabla^2\Psi = -\Psi$, in a D-warped elliptical region with radius,

$$R(\theta) = \pi/2 \times (1 - \epsilon \cos^2(\theta - \phi \sin \theta)), \quad (4.1)$$

and boundary conditions, $y(r=0) = 1, y(r=R(\theta)) = 0$. ie. zero at the boundary.

(i) Circular Boundary

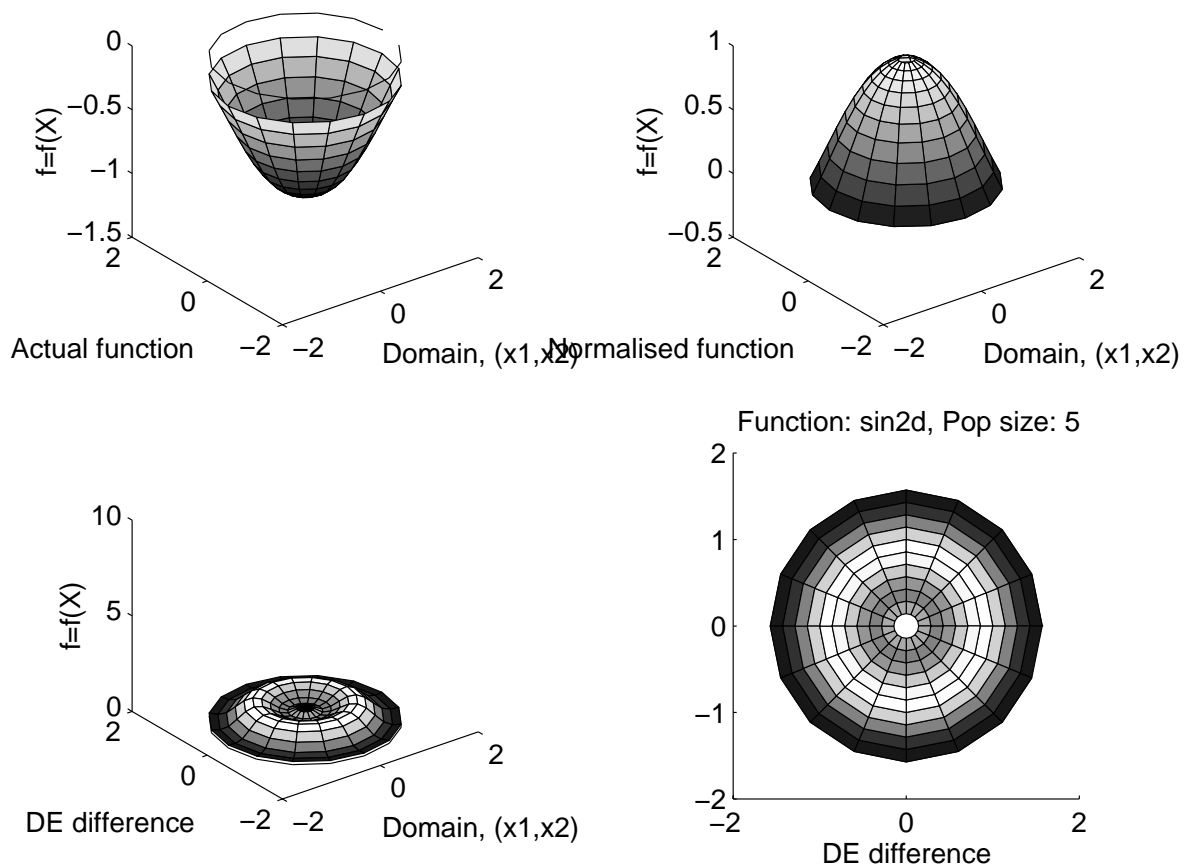


Figure 14. Best individual for circular manifold. Actual result (top left), normalised result (top right), deviation from ideal over range (bottom).

Figure 14 shows the best individual produced for a purely circular manifold ($\epsilon=\phi=0$) on a single run with population 300, 100 generations. The four components to this figure are:

- Top-Left: the *actual* function specified by the best individual
- Top-Right: the same function *normalised* to satisfy the boundary conditions
- Bottom-Left: the difference between LHS and RHS on the normalised fn. ie. $y''(\mathbf{x}) + y(\mathbf{x})$. A correct solution is zero over entire manifold.
- Bottom-Right: top view of same

Figure 15 shows the collated results of 20 runs. The fitness scale used here is calculated from $-\log(-\text{Mean Square Deviation})$, where the 'deviation' is the difference between the LHS and RHS of the differential equation we are trying to solve. This fitness scale should be regarded as a generic logarithmic fitness with larger values indicating a better solution.

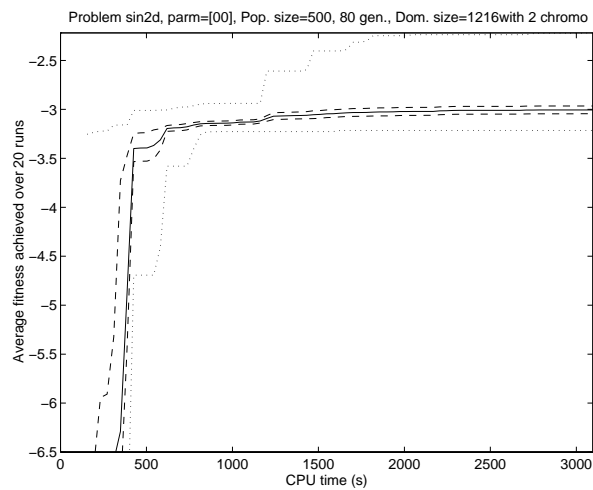


Figure 15. Learning curve for circular manifold.

The best program derived here is equivalent to the expression:

$$y = (-1.289) .* (-1.003) .* \exp((-1.731) .* \exp((3.024) .* ((1.176) .* (-0.474) .* \cos(t1) + (0.2358) .* 1))) ./ ((0.6289) .* \cos((-0.3423) .* 1)),$$

(unnormalised),

where $t1$ is radius, r , and

$t2$ is angle, θ , ie. y is independent of θ .

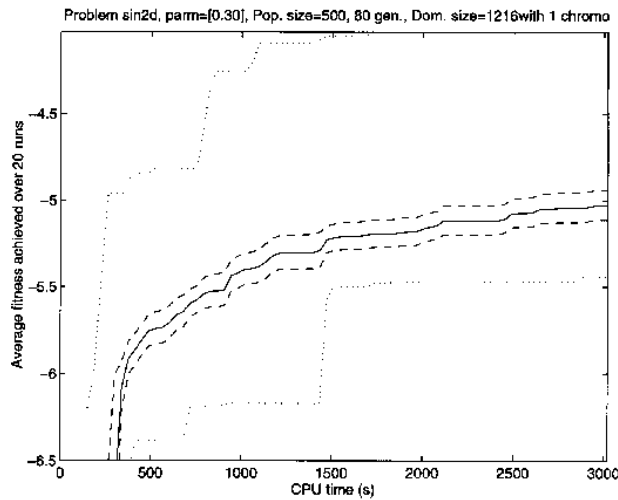


Figure 17. Learning curve for elliptical manifold.

(ii) Non-circular Boundaries

Figure 16 shows the equivalent learning curve for the case of an elliptical boundary ($\epsilon=0.3$, $\phi=0$). It is clear that convergence is much slower for this problem, and that the solution arrived at is of significantly lower quality on average (by a factor of $\sim e^2 \approx 7$). Similarly, Figure 17 shows the learning curve for a D-shaped boundary ($\epsilon=0.4$, $\phi=0.15$). This is another 7 times worse in its mean quality of solution. Note however the extreme variation observed, with the maximum sometimes an order of magnitude fitter than the average.

Thus the quality of the discovered solution is a sensitive function of the degree of complexity inherent in the manifold. This may be partially due to nonlinear effects caused by the normalisation procedure in 2-D on a polar grid. In particular, for the D-

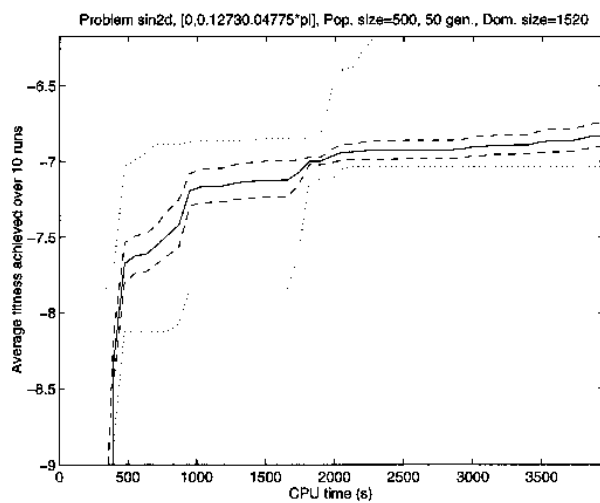


Figure 16. Learning curve for D-shaped manifold.

shaped boundary the maximum of the function may no longer be at the origin ($r=0$). In this case the polar coordinate representation is not as helpful, unless it is somehow recentered on the maximum.

5. Novel Approaches

5.1 Syntax-Preserving Crossover

Many implementations of GP use LISP because of its natural use of recursively defined lists. This allow syntax preserving crossover to be implemented trivially by removing and exchanging entire subtrees as single list elements. The equivalent implementation in RPN exchanges substrings over which the change in the position of the stack-pointer is one. An interesting variation on this is to exchange substrings over which the stack position is preserved. This operation can be interpreted as the exchange of two single-argument functions, but with the arguments left behind. That is, the exchange of two single-argument-function heads. See figures 18 and 19; compare to figures 3 and 4.

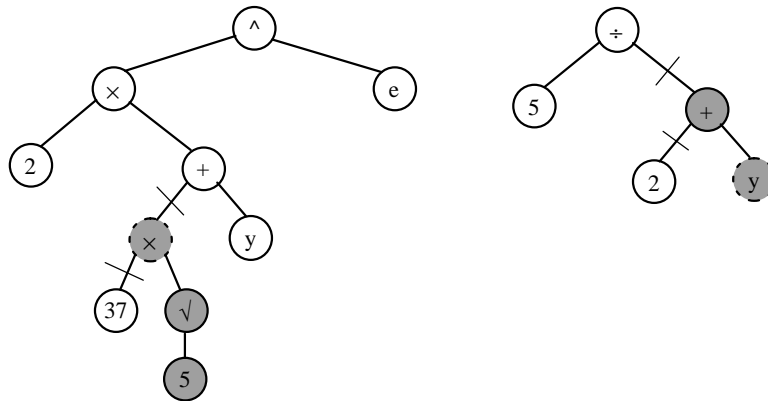


Figure 18. Interpretation of 1-argument functions: $f1(x) = (x \times \sqrt{5}), [5, \sqrt{\cdot}, \times]$; and $f2(x) = (x + y), [y, +]$.

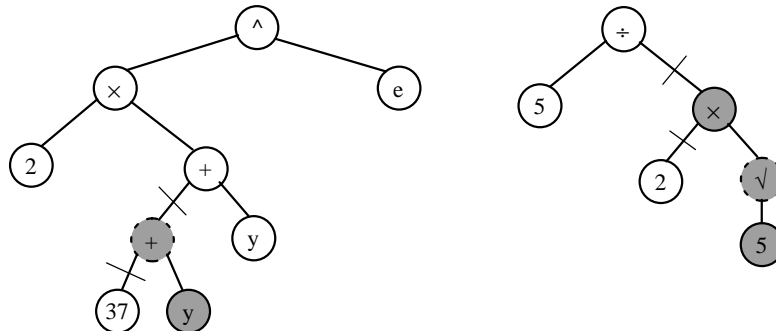


Figure 19. Exchange of RPN subsequences gives new expressions: $(2 \times (37 + y + y))^e, [2, 37, y, +, y, +, \times, e, ^]$; and $5 \div (2 \times \sqrt{5}), [5, 2, \sqrt{\cdot}, \times, \div]$.

In fact the crossover operator for RPN expressions can be easily generalised to define the unambiguous exchange of two n-argument function heads. This new operator still preserves syntax, but it allows a more flexible definition of the composition of the exchanged parts. Several operators from this class (0-arg, 1-arg and 2-arg) were tested on both the 1-D and 2-D sine problems and were discovered to all perform equally well.

5.2 Automatic Definition of Functions

John Koza [6] recommends the Automatic Definition of Functions (ADF) as a way of bootstrapping the acquisition of more complicated functional descriptions. In this approach each individual is partitioned into a main program and a number of sub-routines that may be called from elsewhere in the program. The definition of each of these subroutines *coevolves* with that of the main body. Koza has demonstrated remarkable increases in algorithmic efficiency with this method on a range of problems [6,7]. Unfortunately this approach has the inelegant requirement that the programmer must decide beforehand how many functions will be available, how many arguments they may have, and which functions may call which others (to control recursion).

5.3 Module Acquisition

Perhaps in response to these perceived failings, Angeline has proposed a similar mechanism, called Module Acquisition (MA), that is entirely self-organised [5]. Essentially Module Acquisition allows two new operators that respectively:

- define an arbitrary subtree of code from one individual as an untouchable module that may be called or exchanged by crossover, but not subdivided; and the complementary operator that will
- expand or *unprotect* some module in the population so that it may be further refined.

The modules defined by MA are only available to the rest of the population through reproduction, not by mutation. In evolutionary terms, such a module will only become prevalent if that piece of code is a useful subfunction that should not be further subdivided. If it is not a useful construction then it will be selectively removed from the population as the individuals that contain it die out.

5.4 Continuous Module Acquisition

I have developed a 'continuous' variant of MA by introducing a vector of parameters corresponding roughly to the unnormalised probability that performing crossover on an individual at any particular point will result in an improved fitness. Every time a crossover occurs this so-called 'cut potential' is correspondingly modified: for every point inside the exchanged subtree the potential is reduced by a small fraction; for the node at the root of the subtree the potential is increased by a small amount⁶. Over time the action of evolution should decrease the cut potential at points where crossover has not been helpful and, inevitably, increase it where crossover tends to be most useful. The repeated action of this dynamic establishes and maintains the cut potential vector by the balance of positive and negative feedback.

This Continuous Module Acquisition (CMA) approach was tested and found to successfully define modules or 'genes', ie. to define regions within individuals that have significantly lower cut potential. Unfortunately these automatically-defined genes failed to produce a corresponding increase in efficiency. Figure 20 is an example learning curve for a circular boundary using CMA. Comparison with Figure 15 at 3000s shows no significant difference.

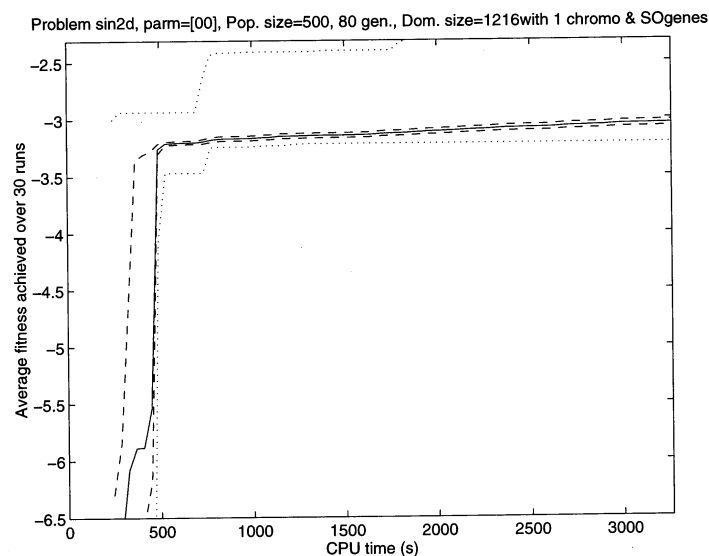


Figure 20. Learning curve with Continuous Module Acquisition.

⁶ There is also a small overall relaxation parameter that ensures that the definition of the vector is a result of recent information, to prevent the process being hijacked by an initial, poor choice. The feedback system used here is very similar to that used by ants (and simulations of ants) to optimise ant-trails.

5.5 Chromosome-Matched Crossover

One significant fact about the implementation of ADFs is that the contents of the *bodies* of the individuals in the population never mix with the contents of the *subroutines*. It was hypothesised that this fact may be the decisive difference between Koza's ADFs and Angeline's MA with regard to the observed superiority of ADFs in [12].

To test this hypothesis I devised an abstraction of Koza's ADF designed to contain only this feature. In this approach the representation of each individual is divided into virtual 'chromosomes' or labelled RPN subsequences. Crossover may exchange genetic material between corresponding chromosomes (ie. having the same label) of two individuals, but not between 'different' chromosomes. Thus the definition of the crossover operator is modified so that any subtree extracted from an individual is truncated at the locations of the, otherwise meaningless, 'chromosome markers', and can only be inserted between the corresponding pair of chromosome markers in another individual. Like ADFs, the number of chromosomes must be specified beforehand (that is unavoidable) but unlike ADFs, no constraint is placed on the form or content of the code in each chromosome. That is, it is not necessary to specify the number of parameters of each function.

This chromosome-matched crossover approach was tested both with and without the previously described CMA, and failed in all cases to present a significant difference in performance. Figure 21 shows an example learning curve for the circular boundary using 2 chromosomes. This may be compared to Figure 15 at 3000s.

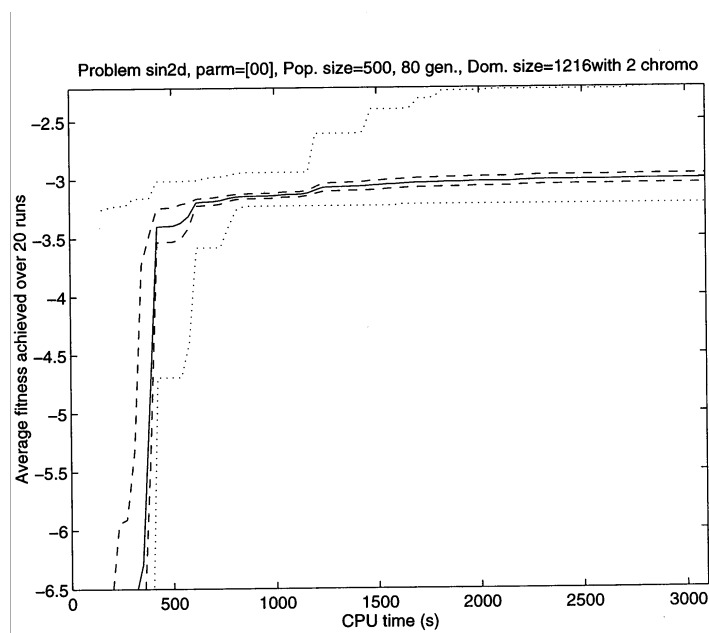


Figure 21. Learning Curve with 2 chromosomes.

6. Discussion

Some comparisons of ADFs to Module Acquisition have shown that for some problems MA has little or no effect while ADF results in considerable improvement [12]. If MA is a less successful approach in general then this may explain the apparent lack of improvement produced by CMA, given its similarity to MA.

Another possibility is that the modules discovered do not reflect any *real* potential for fitter offspring by their preserved existence. Perhaps a better approach would be to adjust the cut potential of the moved nodes by an amount proportional to the increase in fitness (rather than the present fixed adjustment). Still another possibility is that the improvements that occur during evolution in GP occur through the combination of components that are normally incompatible. That is, the potential offspring that are avoided by the CMA process includes some of the most significant improvements.

There are also a number of possible explanations of the failure of chromosome-matched crossover. One is that the crude definition of chromosomes messes up the delicate boundary that effectively defines whether a piece of code can be interpreted as a one or two argument function or as a constant, and this destroys any advantage gained by the segregation of genetic reservoirs. Another possibility is that the observed advantage of ADFs in [12] is not due to the segregation of genetic material.

Many of the expressions produced by GP techniques are complex and opaque and offer no direct insight into the nature of the problem or the solution offered. This is a major criticism of GP techniques as it usually leads to problems concerning the acceptance of the derived programs in industrial applications. Another criticism is that it is difficult to build-in constraints that are already known that may help guide the algorithm to a better solution.

In recent work Marenbach et al. [9] have used a flow-control block representation as the basis for GP to generate dynamic process models. In this approach it is the number, type and relationships between the process blocks that are acted on by the reproduction operators. This approach is a promising one because the derived models can be represented graphically, enabling human users to quickly grasp the relationships and infer meanings, and because known useful constraints or blocks can be built-in or made available to the algorithm for use in its solutions.

A further point to note when considering the results is that the time taken to perform a single run goes approximately as $O(M)$, where M is the number of discrete range/grid/lattice points at which each program is evaluated. This number $M \sim O(L^D)$, where L is the number of subdivisions per dimension and D is the number of dimensions. Thus the total cost increases exponentially with the number of dimensions. A single run for the 1-D case took ~ 10 min on my hardware, while the 2-D case, with significantly fewer subdivisions per dimension, took ~ 1 hr. Recall that 10-

20 runs are required for any confidence in an algorithmic improvement. This stands for a lot of waiting.

7. Conclusions and Further Work

The author believes that further work is justified in the theoretical aspects relating to Automatic Definition of Functions and Module Acquisition. It may also be useful to implement some of the techniques developed in the field to improve properties such as diversity (which was observed to suffer significantly for the more difficult problems), such as pareto analysis and niche techniques.

One approach that might result in significant increases in efficiency is to follow that of Marenbach et al. and divorce the continuous parameter estimation entirely from the genetic algorithm using, for example, gradient descent or simulated annealing to optimise these parameters instead.

The results show that Genetic Programming is an effective technique that can give reasonable results, given plenty of computing resources. In particular, the technique used here can be applied to higher dimensions, however, in practice the algorithmic complexity may be too high.

Several novel approaches gave negative results culminating in a negative result of significant theoretical interest, namely that the syntax-preserving crossover used in Genetic Programming may be generalised to allow the exchange of n-argument functions without adverse effects.

8. Bibliography

- [1] Holland, J. H. (1975). *Adaptation in natural and artificial systems*, Ann Arbor: University of Michigan Press.
- [2] Koza, J. R. (1989). *Hierarchical genetic algorithms operating on populations of computer programs*, Proceedings of the 11th International Joint Conference on Artificial Intelligence, Vol. 1. Morgan Kaufman.
- [3] Horn, J., Goldberg, D. E., Deb, K. (1994). *Long Path Problems*, Parallel Problem Solving from Nature, PPSN III, pp. 149-159, Springer Verlag.
- [4] Cerf, R. *An Asymptotic Theory of Genetic Algorithms*, Lect. Notes in Comp. Sci 1063, Selected Papers, Artificial Evolution AE 95, p 37, Brest, France, Springer.
- [5] Angeline, P. J. (1994). *Genetic Programming and Emergent Intelligence*, Advances in Genetic Programming, Ch. 4, MIT Press.
- [6] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press.
- [7] Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press.
- [8] Culberson, J. C. (1994). *Mutation-Crossover Isomorphisms and the Construction of Discriminating Functions*, Evolutionary Computation Vol. 2/3, MIT Press.
- [9] Marenbach, P., Bettenhausen, K. D., Freyer, S. (1996). *Signal Path Oriented Approach for Generation of Dynamic Process Models*, Proceedings, Genetic Programming 1996, Stamford, CA, USA (GP-96), MIT Press.
- [10] Bäck, T., Hoffmeister, F., Schwefel, H. P. (1991). *A Survey of Evolution Strategies*, Proc. Fourth Int. Conf. on Genetic Algorithms, San Mateo, pp 2-9, Morgan Kaufmann.
- [11] Tuson, A., Ross, P. (1996). *Co-Evolution of Operator Settings in Genetic Algorithms*, Lect. Notes in Comp. Sci., Selected Papers, Evolutionary Computing, AISB Workshop, Brighton UK, Springer.
- [12] Kinnear, K. E. Jr. (1994). *Alternatives in Automatic Definition of Functions: A Comparison of Performance*. Advances in Genetic Programming, Ch. 6, MIT Press.

Appendix A: Program listings

expol.m

```

function [f,x1,x0]=expol(x,t,Nlen);

% EXPOL Expresses an approximate solution to a toy 2nd order diff. eqn. as a fitness value.
%
% function [f,x1,x0]=EXPOL(x,t,Nlen);
%
% t is a vector of ordinates on which x is an approximate solution
%
% Nlen is a vector containing the length of each program. This may
% be used to add a minimum-description-length term to the fitness value.
%
% Also works with x a matrix of row vectors, each approximate solutions.
% Also returns x1 vectors normalised to satisfy boundary conditions, for displaying.
% x0 is the actual solution, if known
%
% If called with two arguments, t=toyl(n,t1), toyl returns an ordinate
% vector, t, of length n, on the interval [0,t1] for evaluation of candidate
% solutions.
%
% Toy problem is x'=x; on interval [0,t1], with x(0)=1, x(t1)=exp(t1);
% Solution is exp(t)
% Fitness is evaluated by direct differentiation

%#inbounds
%#realonly      % must remove to compile with gradient

l0 = 50;        % weighting for ODE condition
l1 = 0.1;       % weighting for x(0) boundary condition
l2 = 0.1;       % weighting for x(pi/2) boundary condition
l3 = 0.04;      % weighting for continuity condition
l4 = 0.04;      % weighting for smoothness condition
l5 = 0.01;      % weighting for trivial-solution penalty term
l6 = 0.02;      % weight for minimum-representation-length term of Fitness
epsilon = 0.0001; % finite addition to every divisor - prevent div. by zero

dimx=size(x);
if min(dimx)==1, x=x(:)'; dimx=size(x); end

f=zeros(dimx(1),1);

if nargin==1, t=pi/2; end % set default interval

if nargin<3,
% f = (rand:x)./x*t; % evenly spaced, random phase
mbintscalar(x)
mbrealvector(t)
f = epsilon*rand + (0:x-1)./(x-1)*t; % evenly spaced, zero phase
else
mbreal(x)
mbrealvector(t)
mbintvector(Nlen)
if max(size(Nlen))==1, Nlen = Nlen * ones(dimx(1),1); end
x0 = exp(t); % actual solution, used for boundary conds. and plotting
x1 = x - repmat(x(:,1),1,dimx(2)); % put x(0)=0
x1 = x0(1)+(x0(length(t))-x0(1))*x1./repmat(x1(:,dimx(2)),1,dimx(2)); % normalise for boundary
cond'ns
tmp = find(isnan(x1));
x1(tmp) = repmat(100,1,length(tmp)); % fudge NaNs for gradient
[G,tmp] = gradient(x1,t,1:dimx(1)); % discard derivative w.r.t. y
% [G2,tmp] = gradient(G,t,1:dimx(1));
f = -(l0*mean( (x1 - G)'.^2)' ... % x'' = -x , the eq'n, normalised to be invariant to boundary
conditions
+ l1*x(:,1).^2 + l2*(x(:,dimx(2))-1).^2 ... % boundary cond'ns

```

```

+ 16*Nlen ); % minimum-description-length term
% + 15./(mean(x1'.^2)' + epsilon) ); % penalty for trivial sol'n
% + 13*max(G.^2) + 14*max(G2.^2) ... % cond'n that x is smooth and contin.
% ); % ./mean( x(i,:).^2 + 0.00001 ); % normalising factor

% remove infinities, NANS from fitness -> low value

fx = find((abs(f)==inf) + isnan(f));
f(fx) = -10000 + rand(size(fx));
nancount = length(fx);
end

```

first.m

```

function [i,j,v]=first(X,t)

% FIRST Returns index of first non-zero element
%
% [i,j,v]=first(X)
% Same syntax as FIND, but only returns first non-zero element, not all.
%
% Also: [i]=first(X,t)
% t is the value that will be returned by i if X contains no non-zeros elements
%
% See also FIND.

% (C) Glenn Burgess, 09/97

if nargin == 1, t=[]; end

if nargin<=1
    tmp = [find(X(:)); t];
    i=tmp(1);
elseif nargin==2
    [tmp1 tmp2] = find(X);
    i=tmp1(1);
    j=tmp2(1)
elseif nargin==3
    [tmp1 tmp2 tmp3] = find(X);
    i=tmp1(1);
    j=tmp2(1)
    v=tmp3(1);
else
    error('Incorrect number of output arguments')
end

```

gpdepth.m

```

function d = gpdepth(PN,S0)

% GPDEPTH Returns tree-depths corresponding to each instruction in a GP program
%
% d = gpdepth(PN)
% PN is a genetic programming individual

dimN=size(PN); % pop_size x (numC+1)
pop_size = dimN(1);
numC = dimN(2)-1;

if min(dimN)~=1,
    d = zeros(pop_size,numC);
    Nlen=PN(:,1);
    for i=1:pop_size
        d(i,1:Nlen(i)) = gpdepth(PN(i,:));
    end
else
    if numC > PN(1), numC =PN(1); end
    N=PN((1:numC)+1);
    inc = ((N>=10) & (N<=19)) - ((N>=0) & (N<=9)); % from definition of operations
    if nargin == 1, S0 = [0]; end
    if numC == 0, d=[];

```

```

else
    d1 = S0(1);
    if inc(numC) <= 0,
        S = [repmat(d1,1,1-inc(numC))+1, S0(2:length(S0))];
    elseif length(S0) == 1,
        S = 0;
    else
        S = S0(2:length(S0));
    end
    d = [gpdepth(PN(1:numC),S) d1];
end
end
end

```

gpdcode.m

```

function Aout=gpdcode(PN,PC)

% GPDCODE      Decode a genetic program into a RPN expression
%
% function Aout=gpdcode(PN,PC)
%
% Gives a text translation of the genetic programming expression specified
% by the list of instruction codes (for gpop), PN, and corresponding list
% of premultiplier constants, PC. The first element of PN is the program length.
% Also works with PN, PC matrices, then each row is a separate program.

dimN=size(PN);

Nlen=PN(:,1);
N=PN(:,2:dimN(2));

for i=1:dimN(1)
    Aout = '...0,1,t2,t1'; % top element in stack
    for j=1:Nlen(i)
        Aout=gpdop(Aout,N(i,j),PC(i,j));
    end
    if dimN(1)>1, disp(Aout), end
end

```

gpdeval.m

```

function [V,x,y,w1,w2,w3,w4,w5,w6,w7,w8,w9]=gpdeval(PN,PC,q)

% GPDEVAL      Decode a genetic program into an equivalent mathematical expression
%
% function V=gpdeval(PN,PC,depth)
%
% Gives a text translation of the genetic programming expression specified
% by the list of instruction codes (for gpop), PN, and corresponding list
% of premultiplier constants, PC. The first element of PN is the program length.
% Also works with PN, PC matrices, then each row is a separate program.
% depth specifies a stack size. Assumes a stack with 4 columns if depth omitted.

dimN=size(PN);

Nlen=PN(:,1);
N=PN(:,2:dimN(2));

if nargin<3, q=4; end % default to stack_size=4

stparm = 'V,x,y';
for i=1:q-2
    eval(['w' num2str(i) '=0;']);
    stparm = [stparm 'w' num2str(i) ];
end

x='t1'; y='t2'; w1='1'; V='0'; % initial contents of stack

for i=1:dimN(1)
    stparm = 'V,x,y';
    for j=1:q-2

```

```

    eval(['w' num2str(j) '=0;']);
    stparam = [stparam 'w' num2str(j) ];
end
x='t1'; y='t2'; w1='1'; V='0'; % initial contents of stack
for j=1:Nlen(i)
    eval(['[' , stparam, ']=gpdop1(N(i,j),PC(i,j),' , stparam, ');' ]);
end
if dimN(1)>1, disp(V), end
end

```

gpdop.m

```
function Aout=gpdop(Ain,instr,premul)
```

```

% GPDOP Decodes genetic programming computation operation into text
%
% Aout=gpdop(Ain,instr,premul)
% Ain is a string expression representing the operations already performed on a
% reverse-polish-notation stack.
% Aout is the representation of the state after the current operation.
%
% premul is an optional scalar post-multiplier for the first element in the stack
% instr is an integer code specifying which arithmetic operation will be performed.
% Available operations are:
%
%   -1   Chromosome boundary marker - NOP
%   0    Pop from top of stack (x) into variable (V),      x -> V
%   1    Add top two elements of stack,                    y+x -> x
%   2    Subtract,                                         y-x -> x
%   3    Element-by-element, multiply,                    y.*x -> x
%   4    Element-by-element, divide,                      y./x -> x
%   5    Rotate stack up one row,                         x->w',y->x',z->y',w->z'
%   10   Push variable on top of stack,                   V -> x'
%   11   Push zeros on stack,                             0 -> x'
%   12   Push ones on stack,                             1 -> x'
%   13   Push imaginary unit on stack,                   i -> x'
%   14   Push (copy) top of stack,                      x -> x'
%   15   Rotate stack down one row,                     x->y',y->z',z->w',w->x'
%   20   Exchange variable with top of stack,           x <-> V
%   21   Exchange top two elements of stack,            x <-> y
%   22   Unary minus,                                    x -> -x
%   23   Square top of stack,                            x -> x.^2
%   24   Unary divide,                                   x -> 1./x
%   25   Null operation                                  {}
%   26   Exponentiate top of stack                      exp(x) -> x
%   27   Natural logarithm of x                        log(x) -> x
%   28   Sine of x                                       sin(x) -> x
%   29   Cosine of x                                    cos(x) -> x
%   30   Tangent of x                                   tan(x) -> x

if instr== -1,      Aout = [Ain ' ,//']; premul = 1;
elseif instr==0,   Aout = [Ain ' ,-> V'];
elseif instr==1,   Aout = [Ain ' ,+'];
elseif instr==2,   Aout = [Ain ' ,-'];
elseif instr==3,   Aout = [Ain ' ,*'];
elseif instr==4,   Aout = [Ain ' ,/']; % +
elseif instr==5,   Aout = [Ain ' ,R->'];
elseif instr==6,   Aout = [Ain ' ,÷'];

elseif instr==10,  Aout = [Ain ' ,V'];
elseif instr==11,  Aout = [Ain ' ,0'];
elseif instr==12,  Aout = [Ain ' ,1'];
elseif instr==13,  Aout = [Ain ' ,i'];
elseif instr==14,  Aout = [Ain ' ,[CR]'];
elseif instr==15,  Aout = [Ain ' ,R<-'];

elseif instr==20,  Aout = [Ain ' ,x<->V'];
elseif instr==21,  Aout = [Ain ' ,x<->y'];
elseif instr==22,  Aout = [Ain ' ,±'];
elseif instr==23,  Aout = [Ain ' ,^2'];
elseif instr==24,  Aout = [Ain ' ,1/x'];
elseif instr==25,  Aout = [Ain ' ,{ }']; premul = 1;
elseif instr==26,  Aout = [Ain ' ,exp'];

```

```

elseif instr==27,      Aout = [Ain ',log'];
elseif instr==28,      Aout = [Ain ',sin'];
elseif instr==29,      Aout = [Ain ',cos'];
elseif instr==30,      Aout = [Ain ',tan'];
end

if nargin==3, Aout=[ Aout ', ' num2str(premul) ',*' ]; end

```

gpeval.m

```

function y=gpeval(PN,PC,s,t)

% GPEVALEvaluate a genetic program on a vector of points
%
% function y=gpeval(PN,PC,s,t)
%
% Evaluates the genetic programming expression specified by the list of
% instruction codes (for gpop), PN, and corresponding list of premultiplier
% constants, PC. The first element of PN is the program length.
% Also works with PN, PC matrices, then each row is a separate program.
%
% s is the number of rows in the fixed-length, RPN stack used by the GP, and
% t is an input vector (or set of vectors) of ordinates at which the GP is evaluated
% Must have s >= 1 + number of vectors in t

% Must recompile after modifying this file.
%   mcc -ir gpeval gpop
%
%#inbounds          % compiler directives
%mbint(PN) % resulting .mexsg is more efficient *without* these assertions!!
%mbintscalar(s)
%mbvector(t)

dimN=size(PN);

Nlen=PN(:,1);
N=PN(:,2:dimN(2));

if min(size(t))==1, t=t(:)'; end

dimt = size(t);

y=zeros(dimN(1),dimt(2));

for i=1:dimN(1)
% initialise variable to 0, stack to [[t]; 1; 0; 0; ... ]
  A=[ zeros(1,dimt(2)); t; ones(1,dimt(2)); zeros(s-1-dimt(1),dimt(2))];
  for j=1:Nlen(i)
    A=gpop(A,N(i,j),PC(i,j));
  end
  y(i,:)=A(1,:); % discard stack, keep result
end

```

gpfindsb.m

```

function [st,fn]=gpfindsb(PN1,sm,index)

% GPFINDSB      Find genetic program substring
%
% [st,fn]=gpfindsb(PN,sm,i)
% PN contains the number of operations in each individual, followed
% by the sequence of GP operations computed by the program.
% sm is the value that stack sums must equal.
% i is an arbitrary index into this sequence
% st and fn are the start and end indices (into the *program* - add
% one to get indices into PN) of the smallest zero
% stack-sum subexpression containing instruction i, found by
% searching backwards or forwards, as required.
%
% If i is omitted, a set of random indices is used.
% If sm is omitted as well it defaults to zero.

```

```

%
% Will also handle PN a matrix and i a vector of indices. PN should
% have the same number of rows as i has elements. ie. it should
% have one program in each row.
%
% [st,fn]=gpfindsb(PN,sm,PP)
% As above, but PP is a matrix of row-vectors specifying the
% probability of choosing a particular index, i, from each program
% Must have size(PP) = size(PN) - [0 1]

%#inbounds
%#realonly

%mbint(PN1)

epsilon = 0.0001;

% ** WARNING ** For some reason PN is transposed in this function.
% This is correct but arbitrary - take note.

PN=PN1';
dimN=size(PN);           % numC x pop_size
pop_size = dimN(2);
numC = dimN(1) - 1;

Nlen=PN(1,:);
N=PN(2:dimN(1),:);

inc = ((N>=10) & (N<=19)) - ((N>=0) & (N<=9));           % from definition of operations
val = cumsum([zeros(1, dimN(2)); inc]);

intv = [ones(1,pop_size); Nlen]; % initialise chromosome boundary variable

if nargin==1, sm=zeros(1,pop_size); end                    % must choose sm

if nargin<=2,                                             % must choose random index
    index=ceil(rand(1,pop_size).*Nlen);                   % an index in the interval [1,Nlen] inclusive
elseif size(index')== size(PN) - [1 0]                  % 'index' is vector of weights
    PP = index';
    valid = PP.*(repmat(Nlen,numC,1)>=repmat([1:numC]',1,pop_size));
    norm = sum(valid);
    cumprob = cumsum(valid); % set up weighted selection scale
    tmp=rand(1,pop_size).*norm; % a real number between 0 and norm
    index = zeros(1,pop_size);
    for i=1:2:pop_size
        bound1 = [0; find(N(:,i)==-1); Nlen(i)+1]; % chromosome boundaries of first parent
        bound2 = [0; find(N(:,i+1)==-1); Nlen(i+1)+1]; % chromosome boundaries of 2nd parent
        index(i) = first(tmp(i)<=cumprob(:,i)); % an index in the interval [1,Nlen] inclusive
        k = last(bound1 < index(i)); % chromosome of crossover
        intv(:,i:i+1) = [bound1(k)+1, bound2(k)+1;
            bound1(k+1)-1, bound2(k+1)-1]; % interval of chromosomes in each parent
        tnorm = sum(PP(bound2(k)+1:bound2(k+1)-1,i));
        tcumprob = cumsum(PP(bound2(k)+1:bound2(k+1)-1,i));
        tmp = rand*tnorm;
        index(i+1) = first(tmp<=tcumprob) + bound2(k);
    end
else error('Third argument must be vector of indices or matrix of cut-probabilities')
end

if max(size(sm)) == 1, sm=repmat(sm,1,pop_size); end

%mbintvector(index)
%mbintvector(sm)

if dimN(2) ~= length(index), PN=PN'; dimN=size(PN); end
if dimN(2) ~= length(index),
    error('Fatal Error: N must have same number of columns as index');
end

st=zeros(1,dimN(2)); fn=zeros(1,dimN(2));
for j=1:dimN(2) % for each program
    i = index(j); % i is initial, random index
    if inc(i,j)>=sm(j) % go up or down?

```

```

    st(j)=i;          % up - two-argument functions, some one-arg. funks
    fn(j) = first( val(1+(i:Nlen(j)),j) <= val(i,j)+sm(j), Nlen(j)+1-i )-1+i;    % finds index
of first 'zero'-sum cut-off point to right of i
    else
    fn(j)=i;          % down - terms (zero-arg. funks), some one-argument funks
    st(j) = last( ( val(1:i,j) <= val(i+1,j)-sm(j) ), 1); % finds index of last possible cutoff to
left
    end
end
st = max(st,intv(1,:));
fn = min(fn,intv(2,:));

% st and fn are indices in the interval [1,Nlen], inclusive, corresponding
% to the start and finish of the segment of program that can be removed
% leaving a stack-sum increase of sm. ie. pn((st:fn)+1) is removed segment,
% if pn=PN(x,:). ie. one program.

```

gpinit.m

```

function [PN,PC,PR,PP]=gpinit(Nset,n,Nlen,chr)

% GPINITInitialises a population of genetic programs
%
% function [PN,PC,PR,PP]=gpinit(Nset,n,Nlen,chr)
%
% Nset is a set of allowed operation codes (integers) for gpop. n is the number of individuals
% in the population. If n>length(Nset) then gpinit ensures that each opcode appears at least
% once.
%
% PN are the population opcodes, PC are the associated premultiplier constants and PR are the
% reproductive parameters. PP are the (initially uniform) cut-probabilities corresponding
% to each subtree location in each program.
%
% Nlen is an optional parameter to specify the initial length of programs, default is 2.
% chr is a second optional parameter specifying the number of chromosomes that each individual
% will be comprised of. Default is 1.

if nargin < 4, chr=1;
    if nargin < 3, Nlen=2;
    end
end

numN=length(Nset);

len = Nlen - (Nlen>1); % len=1 if Nlen<=2, len=Nlen-1 otherwise

list = 1:numN;

if n>=numN
    PN=[repmat(Nlen,n,1) [reshap(Nset(insort(list,rand(numN,len))),numN,len); ...
        reshape(Nset(ceil(rand(n-numN,len)*numN)),n-numN,len) ]];
else
    PN=[repmat(Nlen,n,1) reshape(Nset(ceil(rand(n,len)*numN)),n,len)];
end

for i = 1:chr-1
    PN(:,ceil(Nlen/chr*i)+1) = repmat(-1,n,1);
end

if Nlen>1, PN=[PN zeros(n,1)]; end

% first element of N is number of instructions in list.

PC=randn(size(PN)-[0 1]);
PR=[ones(n,1)*[0.5 0.5 0.04] ones(size(PC)+[0 3])];
% Params are: (1) prob. of crossover (vs. mutation)
% (2) prob. of insertion (vs. exchange) for crossover
% (3) common Gaussian width factor for mutation
% (4:PC+6) individual Gaussian width factors for mutation for P(1:3) and Ci
PP=ones(size(PN)-[0 1]);

```

gpmain.m

(script for 1-D problems - uses gprun2)

```
%function gpmain

%
[best_fit,PN,PC,PR,Fitness,PZ,t,mixmat,fitmat,grop,tlog]=gprun1('sine1',3*pi/2,300,5,30,30,100,0);
% prob='sine1';pint=3*pi/2;pop_size=300;pop_loss=5;domain_size=30;
% graph_period=30;end_generation=100;desired_fit=0;

num_runs=20;

prob='sine1';
pint=3*pi/2;
pop_size=150;
pop_loss=5;
domain_size=30;
graph_period=100;
end_generation=100;
desired_fit=0;

%for domain_size = [70]

fitstat = repmat(desired_fit,num_runs,end_generation);
tstat = repmat(1000,num_runs,end_generation);

for i=1:num_runs
    [best_fit,PN,PC,PR,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]= gprun2(prob,pint,pop_size,
    ...
        pop_loss,domain_size,graph_period,end_generation,desired_fit);
    fitstat(i,:) = best_fit(1:end_generation)';
    tstat(i,:) = tlog(1:end_generation)';
    disp(['Now ending run ' num2str(i)])
end

ctime = (1:end_generation)/end_generation*min(tstat(:,end_generation));
for i=1:num_runs
    fitstat(i,1:100) = interp1(tstat(i,:),fitstat(i,:),ctime)';
end

meanfit = mean(fitstat);
stdfit = std(fitstat)/sqrt(num_runs);          % std of mean

figure(2)
plot(ctime,meanfit,'w-'), hold on
plot(ctime,meanfit+stdfit,'b--')
plot(ctime,meanfit-stdfit,'b--')
plot(ctime,max(fitstat),'y:')
plot(ctime,min(fitstat),'y:'), hold off
axis([0 min(tstat(:,end_generation)) 0 0.16])
ylabel(['Average std. from solution achieved over ' num2str(num_runs) ' runs'])
xlabel('CPU time (s)')
title(['Problem ' prob ', [0,' num2str(pint/pi) '*pi], Pop. size=' ...
    num2str(pop_size) ', ' num2str(end_generation) ' gen., Dom. size=' ...
    num2str(domain_size)])

print

end%
```

gpmain.m

(script for 2-D problems - uses gprun4)

```
%function gpmain

%
[best_fit,PN,PC,PR,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]=gprun2('sine1',3*pi/2,150,5,30,3
0,100,0);
% [best_fit,PN,PC,PR,PP,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]=gprun3('sin2d',[0.3
0.1],100,5,[15 30],5,100,0);
% [best_fit,PN,PC,PR,PP,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]=
%gprun4('sin2d',[0.3 0],[10 14],[300 0.05],[6 8 2],[1.1 0.9 0.002 1 10 2 0],
%['gpmut.m';'gpsexchg';'gpmutate'],10,50);
% prob='sin2d';pint=[0.3 0.1];pop_size=300;pop_loss=15;domain_size=[15 30];
% graph_period=1000;end_generation=1000;desired_fit=0;

desired_fit=0;

num_runs=20;

prob='sin2d';
pint=[0.3 0];
domain_size=[12 16];
pop_parms=[500 0.1];
ind_parms=[6 8 2]; % stack, columns, chromo
alg_parms=[1.1 0.9 0.002 1 10 10 desired_fit]; %fparms(1,2),relax,repnce,mixscle,p_bias,desrd_fit
ropname = ['gpinsrt'; 'gpsexchg'; 'gpmutate'];
graph_period=101;
end_generation=80;

pop_size = pop_parms(1);
pop_loss = pop_parms(2)*pop_size; % number of individuals lost per gen.

fparms = [alg_parms(1) alg_parms(2)]; % [1.1 0.9];
relaxation = alg_parms(3); %0.002;

replace = alg_parms(4); % duplicates are replaced if true
mixscale = alg_parms(5); % scale of mixing perturbation
parent_bias = alg_parms(6); % subtractive handicap on parents' fitness to promote change
desired_fit = alg_parms(7);

stack_size = ind_parms(1); % (6) size of RPN stack
InitCol = ind_parms(2); % (8) initial size of each individual
chromo = ind_parms(3); % (2) number of chromosomes

%for domain_size = [15 30]

fitstat = repmat(desired_fit,num_runs,end_generation);
tstat = repmat(1000,num_runs,end_generation);

for i=1:num_runs
    [best_fit,PN,PC,PR,PP,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]=gprun4(prob,...
    pint,domain_size,pop_parms,ind_parms,alg_parms,ropname,graph_period,end_generation);
    fitstat(i,:) = best_fit(1:end_generation)'; % trace of RMS deviation of 'best individual'
    tstat(i,:) = tlog(1:end_generation)'; % CPU time logged per generation
    disp(['Now ending run ' num2str(i)])
end

% CPU time intervals to resample on
ctime = (1:end_generation)/end_generation*min(tstat(:,end_generation));
% resample fitstat in units of CPU time
for i=1:num_runs
    fitstat(i,1:end_generation) = interp1(tstat(i,:),fitstat(i,:),ctime)';
end

meanfit = mean(fitstat);
stdfit = std(fitstat)/sqrt(num_runs); % std of mean
maxfit=max(fitstat);
minfit=min(fitstat);

figure(6)
plot(ctime,-log(-meanfit),'w-'), hold on
```

```

plot(ctime,-log(-meanfit+stdfit),'b--')
plot(ctime,-log(-meanfit-stdfit),'b--')
plot(ctime,-log(-maxfit),'y:')
plot(ctime,-log(-minfit),'y:'), hold off
axis([0 min(tstat(:,end_generation)) -6.5 -log(-maxfit(end_generation))])
ylabel(['Average fitness achieved over ' num2str(num_runs) ' runs'])
xlabel('CPU time (s)')
title(['Problem ' prob ', parm=[' num2str(pint) '], Pop. size=' ...
num2str(pop_size) ', ' num2str(end_generation) ' gen., Dom. size=' ...
num2str(domain_size) 'with ' num2str(chromo) ' chromo'])

print

fitrange=(-2:0)-ceil(log(-maxfit(end_generation)));
fitprob=zeros(end_generation,length(fitrange));
for i=1:length(fitrange)
% fitprob(:,i)=mean(-log(-fitstat)>fitrange(i));
fitprob(:,i)=nerfc(-exp(-fitrange(i)),stdfit*sqrt(num_runs),meanfit));
end
figure(7)
plot(ctime,fitprob)
ylabel(['Probability of reaching fitness ' num2str(fitrange)])
xlabel('CPU time (s)')
title(['Problem ' prob ', parm=[' num2str(pint) '], Pop. size=' ...
num2str(pop_size) ', ' num2str(end_generation) ' gen., Dom. size=' ...
num2str(domain_size) ' with ' num2str(chromo) ' chromo'])

print

%end

```

gpmut.m

```

function [new_pN, new_pC, new_pR, new_pP]=gpmut(pN,pC,pR,pP,Nset)

% GPMUT Mutate program encodings of two genetic programming individuals
%
% [new_pN, new_pC, new_pR, new_pP]=gpmut(pN,pC,pR,pP,Nset)
% pn,pC,pR,pP each contain two row-vectors of the 'parents' initial parameters.
% Nset is a vector containing all allowed operations
% new_pN, new_pC, new_pR are the mutated versions, respectively.
%
% Changes
%
% Also works for a population, with parents taken from adjacent rows.

%#inbounds
%mbint(pN)

if nargin == 4, Nset = pP; end

dimN=size(pN);
pop_size=dimN(1);
numC=dimN(2)-1; % number of columns

%#mu=reshape(mean(reshape(pR,2)),dimN(1)/2); % mean of parents' parameters (dimN(2)/2 X
numC)
%#sigma=reshape(diff(reshape(pR,2)),dimN(1)/2); % diff. of " " "

%PRuse=pR; %mu; % + randn(size(mu)).*sigma
%Com_width=PRuse(:,3); Spec_width=PRuse(:,4:6);
%PC_width=PRuse(:,6+(1:numC));

% Reproduce - Do Mutation

%new_pR = zeros(size(pR)); % preallocate new_pR to allow -i compilation

% modify premultiplicative constants and reproductive constants
%new_pC = pC .* ( 1 + Com_width*ones(1,numC).*randn(pop_size,numC).* PC_width );
%new_pR(:,1:3) = pR(:,1:3) .* ( 1 + Com_width*ones(1,3).*randn(pop_size,3).* Spec_width );
%new_pR(:,3+(1:numC+3)) = pR(:,3+(1:numC+3)) + Com_width*ones(1,numC+3).*randn(pop_size,numC+3);

```

```

% mutate program operations
new_pN=pN;
Nlen = pN(:,1);
i = ceil(rand(pop_size,1).*Nlen); % vector of indices (into pN(j,:)) of mutating instructions
n = ceil(rand(pop_size,1)*length(Nset)); % vector of indices (into Nset) of new instr.
new_pN(i*pop_size + (1:pop_size)') = Nset(n); % replace ith instruction with nth instr. in
Nset

new_pC = pC;
new_pR = pR;
new_pP = pP;

```

gpmutate.m

```

function [new_pN, new_pC, new_pR, new_pP]=gpmutate(pN,pC,pR,pP)

% GPMUTATE Mutate parameters of two genetic programming individuals
%
% function [new_pN, new_pC, new_pR]=gpmutate(pN,pC,pR)
% pn,pC,pR each contain two row-vectors of the 'parents' initial parameters.
% new_pN, new_pC, new_pR are the mutated versions, respectively.
%
% Also works for a population, with parents taken from adjacent rows.

%#inbounds
%mbint(pN)

dimN=size(pN);
pop_size=dimN(1);
numC=dimN(2)-1; % number of columns

%mu=reshap(mean(reshap(pR,2)),dimN(1)/2); % mean of parents' parameters (dimN(2)/2 X
numC)
%sigma=reshap(diff(reshap(pR,2)),dimN(1)/2); % diff. of " " "

PRuse=pR; %mu; % + randn(size(mu)).*sigma
Com_width=PRuse(:,3); Spec_width=PRuse(:,4:6);
PC_width=PRuse(:,6+(1:numC));

% Reproduce - Do Mutation

new_pR = zeros(size(pR)); % preallocate new_pR to allow -i compilation

% modify premultiplicative constants and reproductive constants
new_pC = pC .* ( 1 + Com_width*ones(1,numC).*randn(pop_size,numC).* PC_width );
new_pR(:,1:3) = pR(:,1:3) .* ( 1 + Com_width*ones(1,3).*randn(pop_size,3).* Spec_width );
new_pR(:,3+(1:numC+3)) = pR(:,3+(1:numC+3)) + Com_width*ones(1,numC+3).*randn(pop_size,numC+3);

% leave program operations and cut-probabilities unchanged
new_pN=pN;
new_pP=pP;

```

gpop.m

```

function Aout=gpop(Ain,instr,premul)

% OP Archetypal genetic programming computation operation
%
% Aout=op(Ain,instr,premul)
% Ain is a two component matrix containing a variable register in the first row,
% and the subsequent rows as a reverse polish notation stack.
%
% premul is an optional scalar pre-multiplier for the first element in the stack
% instr is an integer code specifying which arithmetic operation will be performed.
% Available operations are:
% -1 Chromosome boundary marker - NOP
% 0 Pop from top of stack (x) into variable (V), x -> V
% 1 Add top two elements of stack, y+x -> x
% 2 Subtract, y-x -> x
% 3 Element-by-element, multiply, y.*x -> x
% 4 Element-by-element, divide, y./x -> x

```

```

%      5      Rotate stack up one row,          x->w',y->x',z->y',w->z'
%      6      Modified division                y./(x^2+1) -> x
%     10      Push variable on top of stack,    V -> x'
%     11      Push zeros on stack,              0 -> x'
%     12      Push ones on stack,              1 -> x'
%     13      Push imaginary unit on stack,     i -> x'
%     14      Push (copy) top of stack,         x -> x'
%     15      Rotate stack down one row,        x->y',y->z',z->w',w->x'
%     20      Exchange variable with top of stack, x <-> V
%     21      Exchange top two elements of stack, x <-> y
%     22      Unary minus,                      x -> -x
%     23      Square top of stack,              x -> x.^2
%     24      Unary divide,                     x -> 1./x
%     25      Null operation                     {}
%     26      Exponentiate top of stack         exp(x) -> x
%     27      Protected logarithm of x         log(x) -> x
%     28      Sine of x                         sin(x) -> x
%     29      Cosine of x                       cos(x) -> x
%     30      Tangent of x                      tan(x) -> x

%#inbounds
%mbintvector(instr) % resulting .mexsg is more efficient *without* these assertions!!
%mbvector(premul)

dimA=size(Ain);
epsilon=0.00001;

V=Ain(1,:);
Stack=Ain(2:dimA(1),:);
q=dimA(1)-1; % number of rows in stack (depth)

% if nargin==3, Stack(1,:)=Stack(1,)*premul; end
% premul changed to postmul 9/7/97

x=Stack(1,:);
y=Stack(2,:);

vers=version;
if vers(1)==5, warning(off), end

if instr==-1, premul=1;
elseif instr==0, V=x; Stack(1:q-1,:)=Stack(2:q,:);
elseif instr==1, Stack(1:q-1,:)=[y+x; Stack(3:q,:)];
elseif instr==2, Stack(1:q-1,:)=[y-x; Stack(3:q,:)];
elseif instr==3, Stack(1:q-1,:)=[y.*x; Stack(3:q,:)];
elseif instr==4, Stack(1:q-1,:)=[y./(x+rand*epsilon); Stack(3:q,:)];
elseif instr==5, Stack=[Stack(2:q,:); x];
elseif instr==6, Stack(1:q-1,:)=[y./((x.^2)+0.01); Stack(3:q,:)];

elseif instr==10, Stack=[V; Stack(1:q-1,:)];
elseif instr==11, Stack=[zeros(1,dimA(2)); Stack(1:q-1,:)];
elseif instr==12, Stack=[ones(1,dimA(2)); Stack(1:q-1,:)];
%elseif instr==13, Stack=[i*ones(1,dimA(2)); Stack(1:q-1,:)];
%above line commented to support -r optimisation in compilation GB 7/7/97
elseif instr==14, Stack(2:q,:)=Stack(1:q-1,:);
elseif instr==15, Stack=[Stack(q,:); Stack(1:q-1,:)];

elseif instr==20, Stack(1,:)=V; V=x;
elseif instr==21, Stack(1:2,:)=[y; x];
elseif instr==22, Stack(1,:)=-x;
elseif instr==23, Stack(1,:)=x.^2;
elseif instr==24, Stack(1,:)=1./(x+rand*epsilon);
elseif instr==25, premul=1;
elseif instr==26, Stack(1,:)=exp(x);
elseif instr==27, Stack(1,:)=log(abs(x)+rand*epsilon);
elseif instr==28, Stack(1,:)=sin(x);
elseif instr==29, Stack(1,:)=cos(x);
elseif instr==30, Stack(1,:)=tan(x);
end

if vers(1)==5, warning(on), end

if nargin==3

```



```

chromo = 2; % number of chromosomes
%Nset=[0:6 10:12 14:15 20:27] % allowed operations
Nset=[0:4 6 12 14] % allowed operations
%Nset=[0:4 12 14 28];

mixscale = 10 % scale of mixing perturbation
parent_bias = 0.5 % subtractive handicap on parents' fitness to promote change
rop0 = [0 0.5 0.1 0.4] % initial proportions of reproduction operators used

% fwrite(fid,[rand('seed') pop_size pop_loss stack_size InitCol domain_size Nset], 'int');

% Stopping critereon
%desired_fit = 1000;
%end_generation = 100;

% Available operations are:
% 0 Pop from top of stack (x) into variable (V), x -> V
% 1 Add top two elements of stack, y+x -> x
% 2 Subtract, y-x -> x
% 3 Element-by-element, multiply, y.*x -> x
% 4 Element-by-element, divide, y./x -> x
% 5 Rotate stack up one row, x->w',y->x',z->y',w->z'
% 6 Modified division y./(x^2+1) -> x
% 10 Push variable on top of stack, V -> x'
% 11 Push zeros on stack, 0 -> x'
% 12 Push ones on stack, 1 -> x'
% 13 Push imaginary unit on stack, i -> x'
% 14 Push (copy) top of stack, x -> x'
% 15 Rotate stack down one row, x->y',y->z',z->w',w->x'
% 20 Exchange variable with top of stack, x <-> V
% 21 Exchange top two elements of stack, x <-> y
% 22 Unary minus, x -> -x
% 23 Square top of stack, x -> x.^2
% 24 Unary divide, x -> 1./x
% 25 Null operation {}
% 26 Exponentiate top of stack exp(x) -> x
% 27 Natural logarithm of x log(x) -> x
% 28 Sine of x sin(x) -> x
% 29 Cosine of x cos(x) -> x
% 30 Tangent of x tan(x) -> x

% Set up population for Genetic Programming

[PN,PC,PR]=gpinit(Nset,pop_size,InitCol,chromo); % initialise population for GP

sp=size(PC); % PC is pop_size rows X numC columns
pop_size=sp(1); numC=sp(2);
Nlen=PN(:,1);

% Evaluate population for fitness

t = feval(prob,domain_size,pint); % initialise evaluation ordinates
PZ = gpeval(PN,PC,stack_size,t);
[Fitness,NZ,Z0] = feval(prob,PZ,t,Nlen); % check fitness of solutions; incl. min. descr. length
prtrb = 1:pop_size; % initialise previous perturbation vector

% initialise statistic and loop variables

mixmat = zeros(pop_size, end_generation); % image of mixing built from successive fiti
fitmat = zeros(pop_size, end_generation); % image of fitness increase built from successive
fitnesses
grop = zeros(end_generation,4); % reproduction operator use counter
best_fit = zeros(end_generation+1,1);
tlog = zeros(end_generation+1,1);
tlog(1) = cpu0;
mutlog = zeros(pop_size,end_generation);

generation=1;
best_f = -inf;

% begin evolution loop

while (generation <= end_generation) & ((best_f<desired_fit) | (generation < 10))

```

```

% Sort population into order of decreasing fitness

[Fitness, fiti]=sort(-Fitness);
Fitness = -Fitness;
mixmat(:,generation) = prtrb(fiti)';          % effective unperturbed mixing
fitmat(:,generation) = Fitness;
mutlog(:,generation) = PR(fiti,3);

% Store best results so far

if Fitness(fiti(1))>best_f                    % new best fit
    best_fit(generation) = sqrt(sum((NZ(fiti(1,:))-Z0).^2)/domain_size; % update best fit entry
    % RMS deviation from actual solution
    best_f = Fitness(fiti(1));
    best_pn = PN(fiti(1,:));
    best_pc = PC(fiti(1,:));
    best_pr = PR(fiti(1,:));
    best_pz = PZ(fiti(1,:));
    best_nz = NZ(fiti(1,:));
else                                          % same best fit
    best_fit(generation) = best_fit(generation-1);
    fiti = fiti([pop_size, 1:pop_size-1]); % shift indices down one
    blen = length(best_pc);
    PN(fiti(1),1:blen+1) = best_pn; % insert best fit on top of fiti
    PC(fiti(1),1:blen) = best_pc;
    PR(fiti(1),1:blen+6) = best_pr;
    PZ(fiti(1),:) = best_pz;
    NZ(fiti(1),:) = best_nz;
end

% Introduce perturbation to ordering

[tmp, prtrb] = sort((1:pop_size) + randn(1,pop_size)*mixscale); % perturbation
fiti = fiti(prtrb);          % perturbed descending ordering for Fitness

% Sort population into perturbed ordering
PN=PN(fiti,:); PC=PC(fiti,:); PR=PR(fiti,:); PZ=PZ(fiti,:);
Fitness=Fitness(prtrb); NZ=NZ(fiti,:); % normalised result

% Create replacement individuals (in lieu of deaths)
%
%[PN1,PC1,PR1]=gpinit(Nset,pop_loss,InitCol,chromo);
%PZ1=gpeval(PN1,PC1,stack_size,t); % evaluate solutions
%F1=feval(prob,PZ1,t,InitCol); % determine fitness
%
% Replace dead (least fit) individuals
% Should perhaps re-sort population, but won't bother
%
%PN(pop_size + (1-pop_loss:0),1:InitCol+1) = PN1;
%PC(pop_size + (1-pop_loss:0),1:InitCol) = PC1;
%PR(pop_size + (1-pop_loss:0),1:InitCol+6) = PR1;
%PZ(pop_size + (1-pop_loss:0),:) = PZ1;
%Fitness(pop_size + (1-pop_loss:0)) = F1;

% Replace dead (least fit) individuals
% Should perhaps re-sort population, but won't bother

blen = length(best_pc); % number of columns of best program
PN(pop_size + (1-pop_loss:2:0),1:blen+1) = repmat(best_pn,fix(pop_loss/2+0.5),1);
PC(pop_size + (1-pop_loss:2:0),1:blen) = repmat(best_pc,fix(pop_loss/2+0.5),1);
PR(pop_size + (1-pop_loss:2:0),1:blen+6) = repmat(best_pr,fix(pop_loss/2+0.5),1);
PZ(pop_size + (1-pop_loss:2:0),:) = repmat(best_pz,fix(pop_loss/2+0.5),1);
%Fitness(pop_size + (1-pop_loss:0)) = ;

Nlen=PN(:,1);

% Reproduce - Perform mutation and crossover

[PNinsrt, PCinsrt, PRinsrt]= gpinsrt(PN,PC,PR); % may all be different sizes
[PNexchg, PCexchg, PRexchg]= gpexchg(PN,PC,PR);
[PNmutate, PCmutate, PRmutate]=gpmutate(PN,PC,PR);
%[PNinsrt, PCinsrt, PRinsrt]= gpmut(PN,PC,PR,Nset);

```

```

%[PNexchg, PCexchg, PRexchg]= gpfunc(PN,PC,PR);

% Extend number of columns

dim=[size(PC); size(PCexchg); size(PCinsrt); size(PCmutate)];
numC=max(dim(:,2));
new_PN=zeros(pop_size,numC+1);
new_PC=zeros(pop_size,numC);
new_PR=zeros(pop_size,numC+6);

% Evaluate population for fitness

t=feval(prob,domain_size,pint); % new evaluation domain
PZexchg = gpeval(PNexchg,PCexchg,stack_size,t);
PZinsrt = gpeval(PNinsrt,PCinsrt,stack_size,t);
PZmutate = gpeval(PNmutate,PCmutate,stack_size,t);
[Fexchg,NZexchg] = feval(prob,PZexchg,t,PNexchg(:,1)); % compare fitness of offspring
[Finsrt,NZinsrt] = feval(prob,PZinsrt,t,PNinsrt(:,1));
[Fmutate,NZmutate]= feval(prob,PZmutate,t,PNmutate(:,1));
FT=[reshap(Fitness,2)-parent_bias; reshap(Fexchg,2); reshap(Finsrt,2); reshap(Fmutate,2)];
[dummy, pairind]=max(FT);
pairind=fix((pairind+1)/2); % indices (1-4) of winning pairs

%tmp = rand(pop_size/2,1)*sum(rop0); % generate roulette wheel selection of operator
%crop0 = cumsum(rop0);
%pairind = 1 + (tmp>crop0(1)) + (tmp>crop0(2)) + (tmp>crop0(3));

rop=zeros(1,4);
for i=1:2:pop_size
    if pairind((i+1)/2) == 1
        new_PN(i+[0 1],1:dim(1,2)+1) = PN(i+[0 1],:);
        new_PC(i+[0 1],1:dim(1,2)) = PC(i+[0 1],:);
        new_PR(i+[0 1],1:dim(1,2)+6) = PR(i+[0 1],:);
    % PZ(i+[0 1],:) = PZ(i+[0 1],:); % PZ and Fitness unchanged
    % Fitness(i+[0 1]) = Fitness(i+[0 1]); % no need to recalculate
    elseif pairind((i+1)/2) == 2
        new_PN(i+[0 1],1:dim(2,2)+1) = PNexchg(i+[0 1],:);
        new_PC(i+[0 1],1:dim(2,2)) = PCexchg(i+[0 1],:);
        new_PR(i+[0 1],1:dim(2,2)+6) = PRexchg(i+[0 1],:);
        PZ(i+[0 1],:) = PZexchg(i+[0 1],:);
        Fitness(i+[0 1]) = Fexchg(i+[0 1]);
        NZ(i+[0 1],:) = NZexchg(i+[0 1],:);
    elseif pairind((i+1)/2) == 3
        new_PN(i+[0 1],1:dim(3,2)+1) = PNinsrt(i+[0 1],:);
        new_PC(i+[0 1],1:dim(3,2)) = PCinsrt(i+[0 1],:);
        new_PR(i+[0 1],1:dim(3,2)+6) = PRinsrt(i+[0 1],:);
        PZ(i+[0 1],:) = PZinsrt(i+[0 1],:);
        Fitness(i+[0 1]) = Finsrt(i+[0 1]);
        NZ(i+[0 1],:) = NZinsrt(i+[0 1],:);
    elseif pairind((i+1)/2) == 4
        new_PN(i+[0 1],1:dim(4,2)+1) = PNmutate(i+[0 1],:);
        new_PC(i+[0 1],1:dim(4,2)) = PCmutate(i+[0 1],:);
        new_PR(i+[0 1],1:dim(4,2)+6) = PRmutate(i+[0 1],:);
        PZ(i+[0 1],:) = PZmutate(i+[0 1],:);
        Fitness(i+[0 1]) = Fmutate(i+[0 1]);
        NZ(i+[0 1],:) = NZmutate(i+[0 1],:);
    end
    rop(pairind((i+1)/2)) = rop(pairind((i+1)/2)) + 2;
end

grop(generation,:) = rop;

PN=new_PN;
PC=new_PC;
PR=new_PR;
generation=generation+1;

% Plot solutions

disp(['Ending generation ' num2str(generation-1)])
disp(['Best Fitness = ' num2str(best_f) ', Biggest program = ' num2str(numC)])
disp(['Length of Fittest program = ' num2str(best_pn(1))])

```

```

if (rem(generation,graph_period)==0)
    disp(['Indices of discarded individuals: '])
    fiti(pop_size + (-pop_loss:-1) + 1)'
    disp('Expression for best program:')
    gpdeval(best_pn,best_pc)
    rop

    figure(1)
    plot(t,NZ(1:pop_size*0.5,:),' ':'')
    axis([0,t(length(t)),min(Z0)-1,max(Z0)+1])
    hold on, plot(t,Z0,'b')
    plot(t,best_nz,'w-.'')
    plot(t,NZ(1,:),'r--'), hold off
    xlabel('Domain, t')
    ylabel('x=x(t)')
    title(['Function: ' prob ' , Generation: ' num2str(generation-1) ...
        ', Pop size: ' num2str(pop_size)])

    figure(2), imagesc(mixmat(:,1:generation-1)), colormap(jet)
    title('Mixing since inception')

    if generation > 2,
        figure(4)
        mesh(flipud((fitmat(:,1:generation-1)>-5000).*exp(fitmat(:,1:generation-1)/10)))
        colormap(jet)
        title('Fitness since inception'), colorbar
    end

    figure(3)
    plot(2:generation,grop(1:generation-1,1),'b',2:generation,grop(1:generation-
1,2),'g:',2:generation,grop(1:generation-1,3),'r--',2:generation,grop(1:generation-1,4),'c-.'')
    title('Parents (b-), Exchange (g:), Insert (r--), Mutate (c-.)')

    figure(5)
    plot(best_fit(1:generation-1),'w')
    hold on, plot(mean(fitmat(1:(pop_size*0.5),1:generation-1)),'r:'), hold off
    % axis([0 generation-1 0 1])
    title('Best fitness (solid), Mean fitness of top 50% (dotted)')
    xlabel('Generations')
    drawnow
end % graph_period

tlog(generation) = cputime;

end % while best_fit<desired_fit

% Sort population in order of decreasing fitness

[Fitness, fiti]=sort(-Fitness);
Fitness = -Fitness;
PN=PN(fiti,:); PC=PC(fiti,:); PR=PR(fiti,:); PZ=PZ(fiti,:);
NZ=NZ(fiti,:); % normalised result

%diary off
figure(1)
plot(t,NZ(1:pop_size*0.5,:),'g:')
axis([0,t(length(t)),min(Z0)-1,max(Z0)+1])
hold on, plot(t,Z0,'b')
plot(t,best_nz,'w-.'')
plot(t,NZ(1,:),'r--'), hold off
xlabel('Domain, t')
ylabel('x=x(t)')
title(['Function: ' prob ' , Generation: ' num2str(generation-1) ...
    ', Pop size: ' num2str(pop_size)])

figure(2), imagesc(mixmat(:,1:generation-1)), colormap(jet)
title('Mixing since inception')

if generation > 2,
    figure(4)
    mesh(flipud((fitmat(:,1:generation-1)>-5000).*exp(fitmat(:,1:generation-1)/10)))
    colormap(jet)
    title('Fitness since inception'), colorbar

```

```

end

figure(3)
plot(2:generation,grop(1:generation-1,1),'b',2:generation,grop(1:generation-
1,2),'g:',2:generation,grop(1:generation-1,3),'r--',2:generation,grop(1:generation-1,4),'c-.')
title('Parents (b-), Exchange (g:), Insert (r--), Mutate (c-.)')

figure(5)
plot(best_fit(1:generation-1),'w')
hold on, plot(mean(fitmat(1:(pop_size*0.5)),1:generation-1),'r:'), hold off
% axis([0 generation-1 0 1])
title('Best fitness (solid), Mean fitness of top 50% (dotted)')
xlabel('Generations')

tlog(generation) = cputime;
tlog = tlog - tlog(1);
for i = find(diff(tlog)<0)
    tlog(i+1:end_generation+1) = tlog(i+1:end_generation+1) + tlog(i);
end

```

gprun4.m

(main program for 2-D problems)

```

function [best_fit,PN,PC,PR,PP,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]= ...
    gprun4(prob,pint,domain_size,pop_params,ind_params,alg_params,ropname,graph_period,end_generation)

% [best_fit,PN,PC,PR,PP,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]=gprun4('sin2d',[0.4
0.15],[15 18],[100 0.1],[6 8 1],[1.1 0.9 0.002 0 10 2 0],['gpxinsrt';'gpxexchg';'gpmutate'],2,50);
% [best_fit,PN,PC,PR,PP,Fitness,PZ,t,Z0,mixmat,fitmat,grop,tlog,mutlog]=gprun4('sin2d',[0.3 0],[10
14],[300 0.05],[6 8 2],[1.1 0.9 0.002 1 10 2 0],['gpmut '];'gpxexchg';'gpmutate'],10,50);
% tic;
[best_fit,PN,PC,PR,Fitness,PZ,t,mixmat,fitmat,grop,tlog]=gprun2('expol',2,150,3,20,30,100,0); toc
% prob='sin2d';pint=[0.4 0.15];pop_size=50;pop_loss=2;domain_size=[20 18];
% graph_period=1;end_generation=10;desired_fit=0;

cpu0=cputime;

rand('seed',123456);
randn('seed',123457);
rand('seed',100*sum(clock));
randn('seed',100*sum(clock)+1);

% log output to file
%t=clock; % make unique filename
%fid=fopen([num2str(t(2:5)) '.log'],'wt');
%diary(['../data/' num2str(t(2:5)) '.log']) % log normal output to file

% Evolution parameters

pop_size = pop_params(1);
pop_loss = pop_params(2)*pop_size; % number of individuals lost per gen.

disp(['Fitness function: ' prob ', with parameter ' num2str(pint) '])
disp(['Population Size = ' num2str(pop_size) ])
disp(['Population Loss = ' num2str(pop_loss) ])
disp(['Domain Size = ' num2str(domain_size) ])

fparms = [alg_params(1) alg_params(2)]; % [1.1 0.9];
relaxation = alg_params(3); %0.002;

replace = alg_params(4); % duplicates are replaced if true
mixscale = alg_params(5); % scale of mixing perturbation
parent_bias = alg_params(6); % subtractive handicap on parents' fitness to promote change
desired_fit = alg_params(7);

stack_size = ind_params(1); % (6) size of RPN stack
InitCol = ind_params(2); % (8) initial size of each individual
chromo = ind_params(3); % (2) number of chromosomes

%Nset=[0:6 10:12 14:15 20:30] % allowed operations
%Nset=[0:4 6 12 14] % minimum allowed operations
Nset=[0:4 12 14 26 28:29]; % min plus some

```

```

rop0 = [0 0.5 0.1 0.4] % initial proportions of reproduction operators used

% fwrite(fid,[rand('seed') pop_size pop_loss stack_size InitCol domain_size Nset], 'int');

% Stopping critereon
%desired_fit = 1000;
%end_generation = 100;

% Available operations are:
%      -1      Chromosome boundary marker - NOP
%      0      Pop from top of stack (x) into variable (V),      x -> V
%      1      Add top two elements of stack,                    y+x -> x
%      2      Subtract,                                        y-x -> x
%      3      Element-by-element, multiply,                    y.*x -> x
%      4      Element-by-element, divide,                      y./x -> x
%      5      Rotate stack up one row,                          x->w',y->x',z->y',w->z'
%      6      Modified division                                y./(x^2+1) -> x
%      10     Push variable on top of stack,                    V -> x'
%      11     Push zeros on stack,                              0 -> x'
%      12     Push ones on stack,                               1 -> x'
%      13     Push imaginary unit on stack,                     i -> x'
%      14     Push (copy) top of stack,                         x -> x'
%      15     Rotate stack down one row,                        x->y',y->z',z->w',w->x'
%      20     Exchange variable with top of stack,              x <-> V
%      21     Exchange top two elements of stack,               x <-> y
%      22     Unary minus,                                     x -> -x
%      23     Square top of stack,                             x -> x.^2
%      24     Unary divide,                                    x -> 1./x
%      25     Null operation                                   {}
%      26     Exponentiate top of stack                        exp(x) -> x
%      27     Protected logarithm of x                        log(|x|) -> x
%      28     Sine of x                                        sin(x) -> x
%      29     Cosine of x                                     cos(x) -> x
%      30     Tangent of x                                    tan(x) -> x

% Set up population for Genetic Programming

[PN,PC,PR,PP]=gpinit(Nset,pop_size,InitCol,chromo); % initialise population for GP

sp=size(PC); % PC is pop_size rows X numC columns
pop_size=sp(1); numC=sp(2);
Nlen=PN(:,1);

% Evaluate population for fitness

%t = feval(prob,domain_size,pint); % initialise evaluation ordinates
t = feval(prob,domain_size); % initialise evaluation ordinates
PZ = gpeval(PN,PC,stack_size,t);
[Fitness,NZ] = feval(prob,domain_size,PZ,t,pint,Nlen); % check fitness of solutions; incl. min.
descr. length
% , Z0]
Z0=0;
prtrb = 1:pop_size; % initialise previous perturbation vector

% initialise statistic and loop variables

mixmat = zeros(pop_size, end_generation); % image of mixing built from successive fiti
fitmat = zeros(pop_size, end_generation); % image of fitness increase built from successive
fitnesses
grop = zeros(end_generation,4); % reproduction operator use counter
best_fit = zeros(end_generation+1,1);
tlog = zeros(end_generation+1,1);
tlog(1) = cpu0;
mutlog = zeros(pop_size,end_generation);

% Set-up polar plot variables

nr = domain_size(1);
ntheta = domain_size(2);
rrange = pi/2*(0:nr-1)/(nr-1);
thetarange = (0:ntheta-1)/ntheta*2*pi;
[r,theta] = meshgrid(rrange,thetarange);

```

```

br = pi/2*(1 - pint(1)*cos(thetarange-pint(2)*sin(thetarange)).^2);           % r(theta) -> range of
'boundary'
% Create mask for defined region, exclude boundary
Inmap = r <= (pi/2*(1 - pint(1)*cos(theta-pint(2)*sin(theta)).^2));
Inmap = Inmap./Inmap;           % 0 -> NaN
X=r.*cos(theta);
Y=r.*sin(theta);

generation=1;
best_f = -inf;

% begin evolution loop

while (generation <= end_generation) & ((best_f<desired_fit) | (generation < 10))

% Sort population into order of decreasing fitness

[Fitness, fiti]=sort(-Fitness);
Fitness = -Fitness;
mixmat(:,generation) = prtrb(fiti)';           % effective unperturbed mixing
fitmat(:,generation) = Fitness;
mutlog(:,generation) = PR(fiti,3);

% Store best results so far

if Fitness(fiti(1))>best_f           % new best fit
    best_fit(generation) = best_f;
    % sqrt(sum((NZ(fiti(1),:)-Z0).^2)/domain_size);   % update best fit entry
    % RMS deviation from actual solution
    best_f = Fitness(fiti(1));
    best_pn = PN(fiti(1),:);
    best_pc = PC(fiti(1),:);
    best_pr = PR(fiti(1),:);
    best_pp = PP(fiti(1),:);
    best_pz = PZ(fiti(1),:);
    best_nz = NZ(fiti(1),:);
else           % same best fit
    best_fit(generation) = best_fit(generation-1);
    fiti = fiti([pop_size, 1:pop_size-1]); % shift indices down one
    blen = length(best_pc);
    PN(fiti(1),1:blen+1) = best_pn; % insert best fit on top of fiti
    PC(fiti(1),1:blen) = best_pc;
    PR(fiti(1),1:blen+6) = best_pr;
    PP(fiti(1),1:blen) = best_pp;
    PZ(fiti(1),:) = best_pz;
    NZ(fiti(1),:) = best_nz;
end

if replace,
    % Find repeated programs (PN only, PC not compared)

    tmp = all([PN(fiti(1:pop_size-1),:)==PN(fiti(2:pop_size),:)]');
    ind = [1 find(1-tmp)+1 find(tmp)+1];
    fiti = fiti(ind);           % shift duplicates to tail
end

% Introduce perturbation to ordering

[tmp, prtrb] = sort((1:pop_size) + randn(1,pop_size)*mixscale); % perturbation
fiti = fiti(prtrb);           % perturbed descending ordering for Fitness

% Sort population into perturbed ordering
PN=PN(fiti,:); PC=PC(fiti,:); PR=PR(fiti,:); PP=PP(fiti,:); PZ=PZ(fiti,:);
Fitness=Fitness(prtrb); NZ=NZ(fiti,:); % normalised result

% Create replacement individuals (in lieu of deaths)
%
%[PN1,PC1,PR1]=gpinit(Nset,pop_loss,InitCol,chromo);
%PZ1=gpeval(PN1,PC1,stack_size,t); % evaluate solutions
%F1=feval(prob,PZ1,t,InitCol); % determine fitness
%
% Replace dead (least fit) individuals
% Should perhaps re-sort population, but won't bother

```

```

%
%PN(pop_size + (1-pop_loss:0),1:InitCol+1) = PN1;
%PC(pop_size + (1-pop_loss:0),1:InitCol) = PC1;
%PR(pop_size + (1-pop_loss:0),1:InitCol+6) = PR1;
%PZ(pop_size + (1-pop_loss:0),:) = PZ1;
%Fitness(pop_size + (1-pop_loss:0)) = F1;

% Replace dead (least fit) individuals
% Should perhaps re-sort population, but won't bother

blen = length(best_pc); % number of columns of best program
PN(pop_size + (1-pop_loss:2:0),1:blen+1) = repmat(best_pn,fix(pop_loss/2+0.5),1);
PC(pop_size + (1-pop_loss:2:0),1:blen) = repmat(best_pc,fix(pop_loss/2+0.5),1);
PR(pop_size + (1-pop_loss:2:0),1:blen+6) = repmat(best_pr,fix(pop_loss/2+0.5),1);
PP(pop_size + (1-pop_loss:2:0),1:blen) = repmat(best_pp,fix(pop_loss/2+0.5),1);
PZ(pop_size + (1-pop_loss:2:0),:) = repmat(best_pz,fix(pop_loss/2+0.5),1);
%Fitness(pop_size + (1-pop_loss:0)) = ;

Nlen=PN(:,1);

% Decay cut-probabilites

PP = gprelax(PP,relaxation);

% Reproduce - Perform mutation and crossover

% may all be different sizes
eval(['PNinsrt, PCinsrt, PRinsrt, PPinsrt']=' ropname(1,:) '(PN,PC,PR,PP,fparms);');
eval(['PNexchg, PCexchg, PRexchg, PPexchg']=' ropname(2,:) '(PN,PC,PR,PP,fparms);');
eval(['PNmutate, PCmutate, PRmutate, PPmutate']=' ropname(3,:) '(PN,PC,PR,PP,fparms);');

% Extend number of columns

dim=[size(PC); size(PCexchg); size(PCinsrt); size(PCmutate)];
numC=max(dim(:,2));
new_PN=zeros(pop_size,numC+1);
new_PC=zeros(pop_size,numC);
new_PR=zeros(pop_size,numC+6);
new_PP=zeros(pop_size,numC);

% Evaluate population for fitness

t=feval(prob, domain_size, pint); % new evaluation domain
PZexchg = gpeval(PNexchg, PCexchg, stack_size, t);
PZinsrt = gpeval(PNinsrt, PCinsrt, stack_size, t);
PZmutate = gpeval(PNmutate, PCmutate, stack_size, t);
[Fexchg, NZexchg] = feval(prob, domain_size, PZexchg, t, pint, PNexchg(:,1)); % compare fitness of
offspring
[Finsrt, NZinsrt] = feval(prob, domain_size, PZinsrt, t, pint, PNinsrt(:,1));
[Fmutate, NZmutate] = feval(prob, domain_size, PZmutate, t, pint, PNmutate(:,1));
FT=[reshape(Fitness,2)-parent_bias; reshape(Fexchg,2); reshape(Finsrt,2); reshape(Fmutate,2)];
[dummy, pairind]=max(FT);
pairind=fix((pairind+1)/2); % indices (1-4) of winning pairs

%tmp = rand(pop_size/2,1)*sum(rop0); % generate roulette wheel selection of operator
%crop0 = cumsum(rop0);
%pairind = 1 + (tmp>crop0(1)) + (tmp>crop0(2)) + (tmp>crop0(3));

rop=zeros(1,4);
for i=1:2:pop_size
    if pairind((i+1)/2) == 1
        new_PN(i+[0 1],1:dim(1,2)+1) = PN(i+[0 1],:);
        new_PC(i+[0 1],1:dim(1,2)) = PC(i+[0 1],:);
        new_PR(i+[0 1],1:dim(1,2)+6) = PR(i+[0 1],:);
        new_PP(i+[0 1],1:dim(1,2)) = PP(i+[0 1],:);
    % PZ(i+[0 1],:) = PZ(i+[0 1],:); % PZ and Fitness unchanged
    % Fitness(i+[0 1]) = Fitness(i+[0 1]); % no need to recalculate
    elseif pairind((i+1)/2) == 2
        new_PN(i+[0 1],1:dim(2,2)+1) = PNexchg(i+[0 1],:);
        new_PC(i+[0 1],1:dim(2,2)) = PCexchg(i+[0 1],:);
        new_PR(i+[0 1],1:dim(2,2)+6) = PRexchg(i+[0 1],:);
        new_PP(i+[0 1],1:dim(2,2)) = PPexchg(i+[0 1],:);
        PZ(i+[0 1],:) = PZexchg(i+[0 1],:);

```

```

    Fitness(i+[0 1]) = Fexchg(i+[0 1]);
    NZ(i+[0 1],:) = NZexchg(i+[0 1],:);
elseif pairind((i+1)/2) == 3
    new_PN(i+[0 1],1:dim(3,2)+1) = PNinsrt(i+[0 1],:);
    new_PC(i+[0 1],1:dim(3,2)) = PCinsrt(i+[0 1],:);
    new_PR(i+[0 1],1:dim(3,2)+6) = PRinsrt(i+[0 1],:);
    new_PP(i+[0 1],1:dim(3,2)) = PPinsrt(i+[0 1],:);
    PZ(i+[0 1],:) = PZinsrt(i+[0 1],:);
    Fitness(i+[0 1]) = Finsrt(i+[0 1]);
    NZ(i+[0 1],:) = NZinsrt(i+[0 1],:);
elseif pairind((i+1)/2) == 4
    new_PN(i+[0 1],1:dim(4,2)+1) = PNmutate(i+[0 1],:);
    new_PC(i+[0 1],1:dim(4,2)) = PCmutate(i+[0 1],:);
    new_PR(i+[0 1],1:dim(4,2)+6) = PRmutate(i+[0 1],:);
    new_PP(i+[0 1],1:dim(4,2)) = PPmutate(i+[0 1],:);
    PZ(i+[0 1],:) = PZmutate(i+[0 1],:);
    Fitness(i+[0 1]) = Fmutate(i+[0 1]);
    NZ(i+[0 1],:) = NZmutate(i+[0 1],:);
end
rop(pairind((i+1)/2)) = rop(pairind((i+1)/2)) + 2;
end

grop(generation,:) = rop;

PN=new_PN;
PC=new_PC;
PR=new_PR;
PP=new_PP;
generation=generation+1;

% Plot solutions

disp(['Ending generation ' num2str(generation-1)])
disp(['Best Fitness = ' num2str(best_f) ', Biggest program = ' num2str(numC)])
disp(['Length of Fittest program = ' num2str(best_pn(1))])

if (rem(generation,graph_period)==0)
    disp(['Indices of discarded individuals: '])
    fiti(pop_size + (-pop_loss:-1) + 1)
    disp('Expression for best program:')
    gpdeval(best_pn,best_pc,stack_size)
    rop

    figure(1)
    nz = reshape(best_nz,ntheta,nr);
    surf([X; X(1,:)],[Y; Y(1,:)],[nz.*Inmap; nz(1,:)])
    xlabel('Domain, (x1,x2)')
    ylabel('Best function')
    zlabel('f=f(X)')
    title(['Function: ' prob ' , Generation: ' num2str(generation-1) ...
        ', Pop size: ' num2str(pop_size)])
    colormap('jet')
    axis('square')
    hold on
    plot3(br.*cos(thetarange),br.*sin(thetarange),zeros(size(thetarange)))
    hold off

    figure(2), imagesc(mixmat(:,1:generation-1)), colormap(jet)
    title('Mixing since inception')

    figure(3)
    plot(2:generation,grop(1:generation-1,1),'b',2:generation,grop(1:generation-
1,2),'g',2:generation,grop(1:generation-1,3),'r--',2:generation,grop(1:generation-1,4),'c-.')
    title(['Parents (b-), ' ropname(1,:) ' (g:), ' ropname(2,:) ' (r--), ' ropname(3,:) ' (c-.)'])

    if generation > 2,
        figure(4)
        mesh(-flipud(log(-fitmat(:,1:generation-1)).*(fitmat(:,1:generation-1)>-
5000))./(fitmat(:,1:generation-1)>-5000)))
        colormap(jet)
        title('Fitness since inception'), colorbar
    end
end

```

```

figure(5)
surf([X; X(1,:)],[Y; Y(1,:)],[nz.*Inmap; nz(1,:)])
view([0 0 1])
title(['Function: ' prob ' , Generation: ' num2str(generation-1) ...
      ', Pop size: ' num2str(pop_size)])
colormap('jet')
axis('square')
hold on
plot3(br.*cos(thetarange),br.*sin(thetarange),zeros(size(thetarange)))
hold off

drawnow
end % graph_period

tlog(generation) = cputime;

end % while best_fit<desired_fit

% Sort population in order of decreasing fitness
[Fitness, fiti]=sort(-Fitness);
Fitness = -Fitness;
PN=PN(fiti,:); PC=PC(fiti,:); PR=PR(fiti,:); PZ=PZ(fiti,:);
NZ=NZ(fiti,:); % normalised result

%diary off
disp('Expression for best program:')
gpdeval(best_pn,best_pc,stack_size)

if graph_period<=end_generation,
figure(1)
nz = reshape(best_nz,ntheta,nr);
surf([X; X(1,:)],[Y; Y(1,:)],[nz.*Inmap; nz(1,:)])
xlabel('Domain, (x1,x2)')
ylabel('Best function')
zlabel('f=f(X)')
title(['Function: ' prob ' , Generation: ' num2str(generation-1) ...
      ', Pop size: ' num2str(pop_size)])
axis('square')
hold on
plot3(br.*cos(thetarange),br.*sin(thetarange),zeros(size(thetarange)))
hold off

figure(5)
surf([X; X(1,:)],[Y; Y(1,:)],[nz.*Inmap; nz(1,:)])
view([0 0 1])
title(['Function: ' prob ' , Generation: ' num2str(generation-1) ...
      ', Pop size: ' num2str(pop_size)])
colormap('jet')
axis('square')
hold on
plot3(br.*cos(thetarange),br.*sin(thetarange),zeros(size(thetarange)))
hold off

figure(2), imagesc(mixmat(:,1:generation-1)), colormap(jet)
title('Mixing since inception')

figure(3)
plot(2:generation,grop(1:generation-1,1),'b',2:generation,grop(1:generation-
1,2),'g',2:generation,grop(1:generation-1,3),'r--',2:generation,grop(1:generation-1,4),'c-.')
title(['Parents (b-), ' ropname(1,:) ' (g:), ' ropname(2,:) ' (r--), ' ropname(3,:) ' (c-.)'])

if generation > 2,
figure(4)
% mesh(flipud(exp(fitmat(:,1:generation-1)/100).*(fitmat(:,1:generation-1)>-
5000)./(fitmat(:,1:generation-1)>-5000)))
mesh(-flipud(log(-fitmat(:,1:generation-1)).*(fitmat(:,1:generation-1)>-
5000)./(fitmat(:,1:generation-1)>-5000)))
colormap(jet)
title('Fitness since inception'), colorbar
end

end % plot condition

```

```

tlog(generation) = cputime;

tlog = tlog - tlog(1);
for i = find(diff(tlog)<0)
    tlog(i+1:end_generation+1) = tlog(i+1:end_generation+1) + tlog(i);
end

```

gpxexchg.m

```

function [new_pN, new_pC, new_pR, new_pP]=gpxexchg(pN,pC,pR,st,fn,new_Nlen);

% GPXEXCHG    Perform crossover with exchange on two genetic programming individuals
%
% [new_pN, new_pC, new_pR]=gpxexchg(pN,pC,pR,st,fn,new_Nlen)
% pn,pC,pR each contain two row-vectors of the 'parents' initial parameters.
% new_pN, new_pC, new_pR are the modified versions, respectively.
%
% st,fn,new_Nlen are optionally pre-calculated start and finish indices for segments
% and resulting new lengths of N vectors. Note: this calculation is non-deterministic
% so it should only be done once.
%
% Alternatively,
% [new_pN, new_pC, new_pR, new_pP]=gpxexchg(pN,pC,pR,pP,r)
% pP contains row-vectors of weights corresponding to the probability of selecting
% each particular subtree,
% r is (optionally) a pair of multiplicative factors applied to the probabilities
% of the end and internal points of the inserted subtrees.
%
% Also works for a population, with parents taken from adjacent rows.

%#inbounds
%mbint(pN)

sm = 1;          % value that subtree stack should sum to
dimN=size(pN);

pop_size = dimN(1);
numC = dimN(2);

Nlen=pN(:,1);
N=pN(:,2:numC);

if nargin==3
    [st, fn]=gpfindsb(pN,sm);          % find start and finish of each subtree
elseif nargin==4
    pP = st;
    r = [1 1];
    [st, fn]=gpfindsb(pN,sm,pP);      % find start and finish of each subtree
elseif nargin==5
    pP = st;          % matrix of cut probabilities
    r = fn;          % filter factors
    [st, fn]=gpfindsb(pN,sm,pP);      % find start and finish of each subtree
end

if nargin<6
    st1=reshape(flipud(reshape(st,2)));          % exchange neighbouring elements
    fn1=reshape(flipud(reshape(fn,2)));          % of st, fn
    new_Nlen = Nlen' + (fn1 - st1) - (fn - st);
end

%mbintvector(st)
%mbintvector(fn)
%mbintvector(new_Nlen)

new_numC=max(new_Nlen); % number of columns to make output

mu=reshape(mean(reshape(pR,2)),pop_size/2);      % mean of parents' parameters (dimN(2)/2    X
numC)
sigma=reshape(diff(reshape(pR,2)),pop_size/2);  % diff. of " " "

new_pN=zeros(pop_size,new_numC+1);

```

```

new_pC=zeros(pop_size,new_numC);
new_pR=zeros(pop_size,new_numC+6);
new_pP=zeros(pop_size,new_numC);

% exchange substrings
for i=1:pop_size
    partner = 2*fix((i+1)/2) - rem(i+1,2); % index of other parent
    new_pN(i,1:new_Nlen(i)+1) = [new_Nlen(i), N(i,1:st(i)-1), ...
        N(partner,st(partner):fn(partner)), N(i,fn(i)+1:Nlen(i))];
    new_pC(i,1:new_Nlen(i)) = [pC(i,1:st(i)-1), pC(partner,st(partner):fn(partner)), ...
        pC(i,fn(i)+1:Nlen(i))];
    new_pR(i,1:new_Nlen(i)+6) = [mu(fix((i+1)/2),1:6) + randn(1,6).*sigma(fix((i+1)/2),1:6)/2, ...
        pR(i,6+(1:st(i)-1)), pR(partner,6+(st(partner):fn(partner))), pR(i,6+(fn(i)+1:Nlen(i)))]];
    if nargin==4
        if st(partner)>fn(partner)
            new_pP(i,1:new_Nlen(i)) = [pP(i,1:st(i)-1), pP(i,fn(i)+1:Nlen(i))];
        else
            new_pP(i,1:new_Nlen(i)) = [pP(i,1:st(i)-1), ...
                r(2)*pP(partner,st(partner):fn(partner)-1), r(1)*pP(partner,fn(partner)), ...
                pP(i,fn(i)+1:Nlen(i))];
        end
    else
        if st(partner)==fn(partner)
            new_pP(i,1:new_Nlen(i)) = [pP(i,1:st(i)-1), r(1)*pP(partner,st(partner)), ...
                pP(i,fn(i)+1:Nlen(i))];
        else
            new_pP(i,1:new_Nlen(i)) = [pP(i,1:st(i)-1), r(1)*pP(partner,st(partner)), ...
                r(2)*pP(partner,st(partner)+1:fn(partner)-1), r(1)*pP(partner,fn(partner)), ...
                pP(i,fn(i)+1:Nlen(i))];
        end
    end
    new_pP(i,:) = new_pP(i, :)*(new_Nlen(i)/sum(new_pP(i, :))); % normalise pP
end
end

```

gpxfunc.m

```

function [new_pN, new_pC, new_pR, new_pP]=gpxfunc(pN,pC,pR,st,fn,new_Nlen);

% GPXFUNC Perform crossover with exchange on two genetic programming individuals
%
% [new_pN, new_pC, new_pR]=gpxfunc(pN,pC,pR,st,fn,new_Nlen)
% pn,pC,pR each contain two row-vectors of the 'parents' initial parameters.
% new_pN, new_pC, new_pR are the modified versions, respectively.
%
% st,fn,new_Nlen are optionally pre-calculated start and finish indices for segments
% and resulting new lengths of N vectors. Note: this calculation is non-deterministic
% so it should only be done once.
%
% Alternatively,
% [new_pN, new_pC, new_pR, new_pP]=gpxfunc(pN,pC,pR,pP,r)
% pP contains row-vectors of weights corresponding to the probability of selecting
% each particular subtree,
% r is (optionally) a pair of multiplicative factors applied to the probabilities
% of the end and internal points of the inserted subtrees.
%
% Also works for a population, with parents taken from adjacent rows.

%#inbounds
%mbint(pN)

dimN=size(pN);

pop_size = dimN(1);
numC = dimN(2);

Nlen=pN(:,1);
N=pN(:,2:numC);

sm = floor(rand(1,pop_size)*2.5-0.5); % value that subtree stack should sum to

if nargin==3
    [st, fn]=gpfindsb(pN,sm); % find start and finish of each subtree
elseif nargin==4

```

```

    pP = st;
    r = [1 1];
    [st, fn]=gpfindsb(pN,sm,pP); % find start and finish of each subtree
elseif nargin==5
    pP = st; % matrix of cut probabilities
    r = fn; % filter factors
    [st, fn]=gpfindsb(pN,sm,pP); % find start and finish of each subtree
end

if nargin<6
    st1=reshap(flipud(reshap(st,2))); % exchange neighbouring elements
    fn1=reshap(flipud(reshap(fn,2))); % of st, fn
    new_Nlen = Nlen' + (fn1 - st1) - (fn - st);
end

%mbintvector(st)
%mbintvector(fn)
%mbintvector(new_Nlen)

new_numC=max(new_Nlen); % number of columns to make output

mu=reshap(mean(reshap(pR,2)),pop_size/2); % mean of parents' parameters (dimN(2)/2 X
numC)
sigma=reshap(diff(reshap(pR,2)),pop_size/2); % diff. of " " "

new_pN=zeros(pop_size,new_numC+1);
new_pC=zeros(pop_size,new_numC);
new_pR=zeros(pop_size,new_numC+6);
new_pP=zeros(pop_size,new_numC);

% exchange substrings
for i=1:pop_size
    partner = 2*fix((i+1)/2) - rem(i+1,2); % index of other parent
    new_pN(i,1:new_Nlen(i)+1) = [new_Nlen(i), N(i,1:st(i)-1), ...
        N(partner,st(partner):fn(partner)), N(i,fn(i)+1:Nlen(i))];
    new_pC(i,1:new_Nlen(i)) = [pC(i,1:st(i)-1), pC(partner,st(partner):fn(partner)), ...
        pC(i,fn(i)+1:Nlen(i))];
    new_pR(i,1:new_Nlen(i)+6) = [mu(fix((i+1)/2),1:6) + randn(1,6).*sigma(fix((i+1)/2),1:6)/2, ...
        pR(i,6+(1:st(i)-1)), pR(partner,6+(st(partner):fn(partner))), pR(i,6+(fn(i)+1:Nlen(i))];
    if nargin==4
        if st(partner)==fn(partner)
            new_pP(i,1:new_Nlen(i)) = [pP(i,1:st(i)-1), r(1)*pP(partner,st(partner)), ...
                pP(i,fn(i)+1:Nlen(i))];
        else
            new_pP(i,1:new_Nlen(i)) = [pP(i,1:st(i)-1), r(1)*pP(partner,st(partner)), ...
                r(2)*pP(partner,st(partner)+1:fn(partner)-1), r(1)*pP(partner,fn(partner)), ...
                pP(i,fn(i)+1:Nlen(i))];
        end
    end
    new_pP(i,:) = new_pP(i, :)*(new_Nlen(i)/sum(new_pP(i, :))); % normalise pP
end
end

```

gpxinsrt.m

```

function [new_pN, new_pC, new_pR, new_pP]=gpxinsrt(pN,pC,pR,st,fn,new_Nlen);

% GPXINSRT    Perform crossover with insertion on two genetic programming individuals
%
% function [new_pN, new_pC, new_pR]=gpxinsrt(pN,pC,pR,st,fn,new_Nlen)
% pn,pC,pR each contain two row-vectors of the 'parents' initial parameters.
% new_pN, new_pC, new_pR are the modified versions, respectively.
%
% st,fn,new_Nlen are optionally pre-calculated start and finish indices for segments
% and resulting new lengths of N vectors. Note: this calculation is non-deterministic
% so it should only be done once.
%
% Also works for a population, with parents taken from adjacent rows.

%#inbounds
%mbint(pN)

sm = 1;

dimN=size(pN);

pop_size = dimN(1);
numC = dimN(2);

Nlen=pN(:,1);
N=pN(:,2:dimN(2));

if nargin==3
    [st, fn]=gpfindsb(pN,sm);                    % find start and finish of each subtree
elseif nargin==4
    pP = st;
    r = [1 1];
    [st, fn]=gpfindsb(pN,sm,pP);                % find start and finish of each subtree
elseif nargin==5
    pP = st;                                     % matrix of cut probabilities
    r = fn;                                       % filter factors
    [st, fn]=gpfindsb(pN,sm,pP);                % find start and finish of each subtree
end

if nargin<6
    st1=reshape(flipud(reshape(st,2)))';         % start and finish of partner's subtree
    fn1=reshape(flipud(reshape(fn,2)))';
    new_Nlen = Nlen + (fn1 - st1 + 1);
end

%mbintvector(st)
%mbintvector(fn)
%mbintvector(new_Nlen)

new_numC=max(new_Nlen); % number of columns to make output

mu=reshape(mean(reshape(pR,2)),dimN(1)/2);      % mean of parents' parameters (dimN(2)/2 X numC)
sigma=reshape(diff(reshape(pR,2)),dimN(1)/2);  % diff. of " " "

% choose random insertion points
x=fix(rand(dimN(1),1).*(Nlen+1));

new_pN=zeros(dimN(1),new_numC+1);
new_pC=zeros(dimN(1),new_numC);
new_pR=zeros(dimN(1),new_numC+6);
new_pP=zeros(pop_size,new_numC);

% insert substrings
for i=1:dimN(1)
    partner = 2*fix((i+1)/2) - rem(i+1,2);      % index of other parent
    new_pN(i,1:new_Nlen(i)+1) = [new_Nlen(i), N(i,1:x(i)), ...
        N(partner,st(partner):fn(partner)), N(i,x(i)+1:Nlen(i))];
    new_pC(i,1:new_Nlen(i)) = [pC(i,1:x(i)), pC(partner,st(partner):fn(partner)), ...
        pC(i,x(i)+1:Nlen(i))];
    new_pR(i,1:new_Nlen(i)+6) = [mu(fix((i+1)/2),1:6) + randn(1,6).*sigma(fix((i+1)/2),1:6)/2, ...

```

```

    pR(i,6+(1:x(i))), pR(partner,6+(st(partner):fn(partner))), pR(i,6+(x(i)+1:Nlen(i)))]];
if nargout==4
    if st(partner)<=fn(partner)
        new_pP(i,1:new_Nlen(i)) = [pP(i,1:x(i)), ...
            r(2)*pP(partner,st(partner):fn(partner)-1), r(1)*pP(partner,fn(partner)), ...
            pP(i,x(i)+1:Nlen(i))];
    end
%   if st(partner)==fn(partner)
%       new_pP(i,1:new_Nlen(i)) = [pP(i,1:x(i)), r(1)*pP(partner,st(partner)), ...
%           pP(i,x(i)+1:Nlen(i))];
%   else
%       new_pP(i,1:new_Nlen(i)) = [pP(i,1:x(i)), r(1)*pP(partner,st(partner)), ...
%           r(2)*pP(partner,st(partner)+1:fn(partner)-1), r(1)*pP(partner,fn(partner)), ...
%           pP(i,x(i)+1:Nlen(i))];
%   end
    new_pP(i,:) = new_pP(i,)*(new_Nlen(i)/sum(new_pP(i,:))); % normalise pP
end
end

```

insort.m

```

function [Y,i]=insort(X,k)

% Y=insort(X,k)
%
% Sorts the columns of X according to key vector k
% That is arranges the columns of X so that the corresponding
% vector k are in ascending order.
%
% Y returns the ordered columns of X
%
% [Y,i]=insort(X,k) also returns the sorted elements of k
% in vector i

sx=size(X);

[i,ind]=sort(k);

if sx(2)==1, Y=X(ind);
else Y=X(:,ind);
end

```

last.m

```

function [i,j,v]=last(X,t)

% LAST Returns index of last non-zero element
%
% [i,j,v]=last(X)
% Same syntax as FIND, but only returns last non-zero element, not all.
%
% Also: [i]=last(X,t)
% t is the value that will be returned by i if X contains no non-zeros elements
%
% See also FIND.

% (C) Glenn Burgess, 10/97

if nargin == 1, t=[]; end

if nargin<=1
    tmp = [t; find(X(:))];
    i=tmp(length(tmp));
elseif nargin==2
    [tmp1 tmp2] = find(X);
    l=length(tmp1);
    i=tmp1(l);
    j=tmp2(l)
elseif nargin==3
    [tmp1 tmp2 tmp3] = find(X);
    l=length(tmp1);

```

```

    i=tmp1(1);
    j=tmp2(1);
    v=tmp3(1);
else
    error('Incorrect number of output arguments')
end

```

meanmask.m

```

function V=meanmask(R,M)

% MEANMASK    Returns the mean of a matrix over a corresponding mask
%
% function V=meanmask(R,M)
%
% V returns the mean of each column of R masked by the matrix M.
% ie. the sum of each column multiplied by the corresponding column of M,
% divided by the number of non-zero elements in that column of M

% C 9.1997, Glenn Burgess

V = sum(R.*M)./sum(M);

```

nerf.m

```

function p=nerf(x,sigma,mu)

% NERF    Natural error function; for use in theory of errors
%
% p=nerf(x)
% Gives the value of the integral from 0 to x of 1/sqrt(2*pi)*exp(-x^2/2)
% Equivalent to integral from 0 to x of 1/sqrt(2*pi)/sigma*exp(-(x/sigma)^2/2),
% with sigma = 1.
%
% Gives the probability of measuring 0 <= a value <= x from a normal distribution.
% Multiply by 2 to get " " abs(value) <= x.
%
% Also, p=nerf(x,sigma),
% or    p=nerf(x,sigma,mu)
%
% See also, NERFC, ERF.

% (C) 1997, Glenn Burgess

if nargin == 3, x=x-mu; end
mu=0;
if nargin >= 2, x=x./sigma; end
sigma=1;

p = erf(x*sqrt(0.5))*0.5;

```

sin2d.m

```

function [f,X0]=sin2d(domain,x,t,e,Nlen);

% SIN2D Expresses an approximate solution to a 2-D 2nd order diff. eqn. as a fitness value.
%
% function [f,x1,x0]=sin2d(domain,x,t,e,Nlen);
%
% t is a matrix of coordinates (2xN) on which x is an approximate solution
% e is a parameter between 0 and 1 controlling the eccentricity of the
% elliptical region of interest (0 is circular)
% domain is a two element vector containing the dimensions of the grid for
% which t is the matrix of coordinates
%
% Nlen is a vector containing the length of each program. This may
% be used to add a minimum-description-length term to the fitness value.
%
% Also works with x a matrix of row vectors, each approximate solutions.
% Also returns x1 vectors normalised to satisfy boundary conditions, for displaying.
% x0 is the actual solution, if known

```

```

%
% If called with two arguments, t=toyl(nr,ntheta), toyl returns a coordinate
% vector, t, of length nr*ntheta, for evaluation of candidate solutions (in
% polar coordinates) in a circular region of radius pi/2.
%
% Problem is x'=-x;
% in a D-shaped elliptical region with radius:
%   r(theta) = pi/2*(1 - e(1)*cos(theta-e(2)*sin(theta)).^2),
% with boundary conditions: x(R=0)=1, x(R=r(theta))=0;
% Solution is cos(R), for e=[0 0]
%
% Fitness is evaluated by direct differentiation

%#inbounds
%#realonly      % must remove to compile with gradient

l0 = 50;        % weighting for ODE condition
l1 = 0.1;       % weighting for x(0) boundary condition
l2 = 0.1;       % weighting for x(pi/2) boundary condition
l3 = 0.04;      % weighting for continuity condition
l4 = 0.04;      % weighting for smoothness condition
l5 = 0.01;      % weighting for trivial-solution penalty term
l6 = 0.01;      % weight for minimum-representation-length term of Fitness
l7 = 0.02;      % penalty for f(r=0) = 0, to stop cheating on normalisation
epsilon = 0.0001; % finite addition to every divisor - prevent div. by zero
R0 = pi/2;

nr = domain(1);
ntheta = domain(2);
rrange = R0*(0:nr-1)/(nr-1) + rand*epsilon;
thetarange = (0:ntheta-1)/ntheta*2*pi + rand*epsilon;
[r,theta] = meshgrid(rrange,thetarange);

tmp=version;
if tmp(1) == '5', method='*linear';
else method='linear';
end

if nargin<3,
    f = [r(:) theta(:)']; % evenly spaced, zero phase
else
    dimx=size(x);
    if min(dimx)==1, x=x(:)'; dimx=size(x); end

    pop_size = dimx(1);
    f=zeros(pop_size,1);

    if max(size(Nlen))==1, Nlen = Nlen * ones(pop_size,1); end
    br = R0*(1 - e(1)*cos(thetarange-e(2)*sin(thetarange)).^2); % r(theta) -> range of 'boundary'
    % Create mask for defined region, exclude boundary
    Inmap = (r <= (epsilon + R0*(1 - e(1)*cos(theta-e(2)*sin(theta)).^2))).*(r>epsilon);

    % Preallocate Full gradient arrays, ..
    Gr0 = zeros(pop_size,prod(domain));
    Gr20 = Gr0;
    Grtheta20 = Gr0;
    % .. renormalised function, ..
    X0 = Gr0;
    % and boundary arrays.
    XX0 = zeros(pop_size,ntheta);
    XX1 = XX0;

    for j = 1:pop_size
        x2 = reshape(x(j,:),ntheta,nr); % make each function an image again
        xx = zeros(1,ntheta);
        for i = 1:ntheta
            tmp = first(rrange>br(i));
            xx(i) = x2(i,tmp) + (x2(i,tmp-1)-x2(i,tmp))/(rrange(1)-rrange(2))*(br(i)-rrange(tmp));
        %   xx(i) = interp1(rrange,x2(i,:),br(i),method); % value of x(theta) at boundary
        end
        %   x1 = x2 - repmat(xx',1,nr); % normalise for 1st boundary cond'n (x(r(theta))=0)
        %   x1 = x1./repmat(x1(:,1),1,nr);% normalise for 2nd boundary cond'n (x(0)=1)
        x1 = (x2 - repmat(xx',1,nr)).*r./repmat(br',1,nr)./repmat(x2(:,1),1,nr);
    end

```

```

tmp = find(isnan(x1));
x1(tmp) = 100*rand(1,length(tmp)); % fudge NANS for gradient

[Gr,Gtheta] = gradient(x1,R0/(nr-1),2*pi/ntheta); % find derivatives w.r.t. r and theta
[Gr2,tmp] = gradient(Gr,R0/(nr-1),2*pi/ntheta);
[tmp,Gtheta2] = gradient(Gtheta,R0/(nr-1),2*pi/ntheta);
XX0(j,:) = xx; % value at boundary
XX1(j,:) = x2(:,1)'; % value at range 0
X0(j,:) = x1(:)'; % unwrapped entire function
Gr0(j,:) = Gr(:)'; % unwrapped df/dr
Gr20(j,:) = Gr2(:)'; % unwrapped d2f/dr2
Gtheta20(j,:) = Gtheta2(:)'; % unwrapped d2f/dtheta2
end
f = -(10*meanmask( (X0 + Gr20 + Gr0./repmat(r(:)',pop_size,1) + ...
Gtheta20./repmat(r(:)',pop_size,1).^2',repmat(Inmap(:)',pop_size,1))'.^2 ... % x'' = -x ,
the eq'n, normalised to be invariant to boundary conditions
+ 11*mean((XX1 - 1).^2)' + 12*mean(XX0'.^2)' ... % boundary cond'ns
+ 17*mean(1./XX1'.^2)' ... % f(r=0) ~= 0
+ 16*Nlen.^2 ); % ... % minimum-description-length term
% + mean(15./(x1.^2 + epsilon)') ); % penalty for trivial sol'n
% + 13*max(G.^2) + 14*max(G2.^2) ... % cond'n that x is smooth and contin.
% ); % ./mean( x(i,:).^2 + 0.00001 ); % normalising factor

% remove infinities, NANS from fitness -> low value

fx = find((abs(f)==inf) + isnan(f));
f(fx) = -10000 + rand(size(fx));
nancount = length(fx);
end

```

sine1.m

```

function [f,x1,x0]=sine1(x,t,Nlen);

% SINE1 Expresses an approximate solution to a toy 2nd order diff. eqn. as a fitness value.
%
% function [f,x1,x0]=sine1(x,t,Nlen);
%
% t is a vector of ordinates on which x is an approximate solution
%
% Nlen is a vector containing the length of each program. This may
% be used to add a minimum-description-length term to the fitness value.
%
% Also works with x a matrix of row vectors, each approximate solutions.
% Also returns x1 vectors normalised to satisfy boundary conditions, for displaying.
% x0 is the actual solution, if known
%
% If called with two arguments, t=toyl(n,t1), toyl returns an ordinate
% vector, t, of length n, on the interval [0,t1] for evaluation of candidate
% solutions.
%
% Toy problem is x''=-x; on interval [0,t1], with x(0)=0, x(t1)=sin(t1);
% Solution is sin(t)
% Fitness is evaluated by direct differentiation

%#inbounds
%#realonly % must remove to compile with gradient

l0 = 50; % weighting for ODE condition
l1 = 0.1; % weighting for x(0) boundary condition
l2 = 0.1; % weighting for x(pi/2) boundary condition
l3 = 0.04; % weighting for continuity condition
l4 = 0.04; % weighting for smoothness condition
l5 = 0.01; % weighting for trivial-solution penalty term
l6 = 0.02; % weight for minimum-representation-length term of Fitness
epsilon = 0.0001; % finite addition to every divisor - prevent div. by zero

dimx=size(x);
if min(dimx)==1, x=x(:)'; dimx=size(x); end

f=zeros(dimx(1),1);

```

```

if nargin==1, t=pi/2; end % set default interval

if nargin<3,
% f = (rand:x)./x*t; % evenly spaced, random phase
mbintscalar(x)
mbrealvector(t)
f = epsilon + (0:x-1)./(x-1)*t; % evenly spaced, zero phase
else
mbreal(x)
mbrealvector(t)
mbintvector(Nlen)
if max(size(Nlen))==1, Nlen = Nlen * ones(dimx(1),1); end
x0 = sin(t);
x1 = x - repmat(x(:,1),1,dimx(2)); % normalise for 1st boundary cond'n (x(0)=0)
x1 = x0(length(t))*x1./repmat(x1(:,dimx(2)),1,dimx(2));% normalise for 2nd boundary cond'n
(x(pi/2)=1)
tmp = find(isnan(x1));
x1(tmp) = repmat(100,1,length(tmp)); % fudge NaNs for gradient
[G,tmp] = gradient(x1,t,1:dimx(1)); % discard derivative w.r.t. y
[G2,tmp] = gradient(G,t,1:dimx(1));
f = -(10*mean( (x1 + G2).^2)' ... % x'' = -x , the eq'n, normalised to be invariant to boundary
conditions
+ 11*x(:,1).^2 + 12*(x(:,dimx(2))-x0(length(t))).^2 ... % boundary cond'ns
+ 16*Nlen ... % minimum-description-length term
+ mean(15./(x1.^2 + epsilon)')' ); % penalty for trivial sol'n
% + 13*max(G.^2) + 14*max(G2.^2) ... % cond'n that x is smooth and contin.
% ); %./mean( x(i,:).^2 + 0.00001 );% normalising factor

% remove infinities, NaNs from fitness -> low value

fx = find((abs(f)==inf) + isnan(f));
f(fx) = -10000 + rand(size(fx));
nancount = length(fx);
end

```

