Int. J. Elect. Enging. Educ., Vol. 36, pp. 139–153. Manchester U.P., 1999. Printed in Great Britain

TEACHING GENETIC ALGORITHM USING MATLAB

Y. J. CAO[1] *and Q. H. WU*[2]
[1] *Intelligent Computer Systems Centre, University of the West of England, Bristol, UK*
[2] *Department of Electrical Engineering and Electronics, University of Liverpool, Liverpool, UK*

ABSTRACT In this paper, an attractive approach for teaching genetic algorithm (GA) is presented. This approach is based primarily on using MATLAB in implementing the genetic operators: crossover, mutation and selection. A detailed illustrative example is presented to demonstrate that GA is capable of finding global or near-global optimum solutions of multi-modal functions. An application of GA in designing a robust controller for uncertain control systems is also given to show its potential in designing engineering intelligent systems.

KEYWORDS control systems; genetic algorithm; MATLAB

1 INTRODUCTION

Genetic Algorithm (GA) is a major topic in a neural and evolutionary computing under-graduate (advanced level)/postgraduate course. The course plays an important role in designing intelligent control systems, which is now very attractive and stimulating to students in Electrical Engineering Departments. The usual way to teach genetic algorithm is by the use of PASCAL, C and C++ languages. In this paper, we present an attractive and easy way for teaching such a topic using MATLAB[1]. It has been found that this approach is quite acceptable to the students. Not only can they access the software package conveniently, but also MATLAB provides many toolboxes to support an interactive environment for modelling and simulating a wide variety of dynamic systems including linear, nonlinear discrete-time, continuous-time and hybrid systems. Together with SIMULINK, it also provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations.

The rest of the paper is organised as follows. Section 2 introduces a simple version of genetic algorithm, canonical genetic algorithm (CGA), which was originally developed by Holland[2]. Section 3 describes the implementation details of the genetic operators: crossover, mutation and selection. Section 4 demonstrates the performance of the CGA with a multi-modal function. Section 5 provides an application of the developed genetic operators in designing a robust controller for uncertain plants, and section 6 gives a conclusion.

2 CANONICAL GENETIC ALGORITHM

Genetic algorithms are adaptive algorithms for finding the global optimum solution for an optimization problem. The canonical genetic algorithm developed by Holland is characterised by binary representation of individual solutions, simple problem-independent crossover and mutation operators, and a proportional selection rule. To understand these concepts, consider the standard procedure of the CGA outlined in Fig. 1. The population members are *strings* or *chromosomes*, which as originally conceived are binary representations of solution vectors. CGA undertakes to select subsets (usually pairs) of solutions from a population, called parents, to combine them to produce new solutions called children (or offspring). Rules of combination to yield children are based on the genetic notion of *crossover*, which consists of interchanging solution values of particular variables, together with occasional operations such as random value changes (called *mutations*). Children produced by the mating of parents, and that pass a survivability test, are then available to be chosen as parents of the next generation. The choice of parents to be matched in each generation is based on a biased random sampling scheme, which in some (nonstandard) cases is carried out in parallel over separate subpopulations whose best members are periodically exchanged or shared. The key parts in Fig. 1 are described in detail as follows.

2.1 *Initialisation*

In the initialisation, the first thing to do is to decide the coding structure. Coding for a solution, termed a *chromosome* in GA literature, is usually described as a string of symbols from $\{0, 1\}$. These components of the chromosome are then labeled as *genes*. The number of bits that must be used to describe the parameters is problem dependent. Let each solution in the population of m such solutions x_i, $i = 1, 2, \ldots, m$, be a string of symbols $\{0, 1\}$ of length l. Typically, the initial population of m solutions is selected completely at random, with each bit of each solution having a 50% chance of taking the value 0.

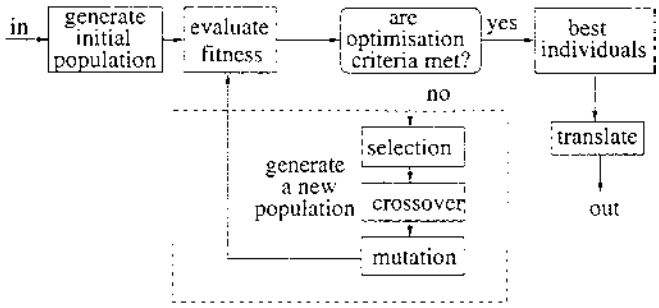


FIG. 1 Standard procedure of a canonical genetic algorithm.

2.2 *Selection*

CGA uses *proportional selection*, the population of the next generation is determined by n independent random experiments; the probability that individual x_i is selected from the tuple $(x_1, x_2, ..., x_m)$ to be a member of the next generation at each experiment is given by

$$P\{x_i \text{ is selected}\} = \frac{f(x_i)}{\sum_{j=1}^{m} f(x_j)} > 0. \tag{1}$$

This process is also called *roulette wheel parent selection* and may be viewed as a roulette wheel where each member of the population is represented by a slice that is directly proportional to the member's fitness. A selection step is then a spin of the wheel, which in the long run tends to eliminate the least fit population members.

2.3 *Crossover*

Crossover is an important random operator in CGA and the function of the crossover operator is to generate new or 'child' chromosomes from two 'parent' chromosomes by combining the information extracted from the parents. The method of crossover used in CGA is the one-point crossover as shown in Fig. 2(a). By this method, for a chromosome of a length l, a random number c between 1 and l is first generated. The first child chromosome is formed by appending the last $l-c$ elements of the first parent chromosome to the first c elements of the second parent chromosome. The second child chromosome is formed by appending the last $l-c$ elements of the second parent chromosome to the first c elements of the first parent chromosome. Typically, the probability for crossover ranges from 0.6 to 0.95.

2.4 *Mutation*

Mutation is another important component in CGA, though it is usually conceived as a background operator. It operates independently on each individual

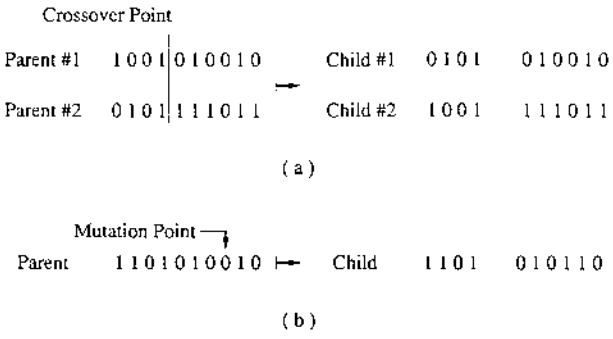


FIG. 2 One-point crossover and mutation operators.

by probabilistically perturbing each bit string. A usual way to mutate used in CGA is to generate a random number v between 1 and l and then make a random change in the vth element of the string with probability $p_m \in (0, 1)$, which is shown in Fig. 2(b). Typically, the probability for bit mutation ranges from 0.001 to 0.01.

Holland's early designs for CGA were simple, but were reported to be effective in solving a number of problems considered to be difficult at the time. The field of genetic algorithms has since evolved, chiefly as a result of innovations in the 1980s, to incorporate more elaborate designs aimed at solving problems in a wide range of practical settings. Characteristically, GAs are distinguished in the following features: (1) they work with a coding of the parameter sets instead of the parameters themselves; (2) they search with a population of points, not a single point; (3) they use the objective function information directly, rather than the derivatives or other auxiliary knowledge, to find maxima; (4) they process information using probabilistic transition rules, rather than deterministic rules. These features make GAs robust to computation, readily implemented with parallel processing and powerful for global optimisation.

3 IMPLEMENTATION OF CGA USING MATLAB

This section describes the procedure of implementing the canonical genetic algorithm using MATLAB. For simplicity, we assume that the task is to achieve the maximum of a one-variable multi-modal function, $f(x) > 0$, for x belonging to the domain $[a, b]$. The implementation steps are as follows.

3.1 *Initialisation*

For CGA, a binary representation is needed to describe each individual in the population of interest. Each individual is made up of a sequence of binary bits (0 and 1). Let *stringlength* and *popsize* denote the length of the binary sequence and the number of individuals involved in the population. Each individual uses a string codification of the form shown in Fig. 3. Using MATLAB, the whole data structure of the population is implemented by a matrix of size *popsize* $\times$ (*stringlength* + 2):

$$\text{pop} = \begin{pmatrix} \text{binary string 1} & x_1 & f(x_1) \\ \text{binary string 2} & x_2 & f(x_2) \\ \vdots & \vdots & \vdots \\ \text{binary string } popsize & x_{popsize} & f(x_{popsize}) \end{pmatrix}$$

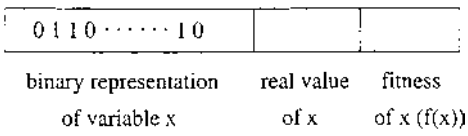


FIG. 3 Binary string representation for the optimisation of a one-variable function.

The first *stringlength* column contains the bits which characterise the binary codification of the real variable x. The strings are randomly generated, but a test must be made to ensure that the corresponding values belong to the function domain. The crossover and mutation operators will be applied on this *stringlength*-bit sub-string. The (*stringlength* + 1)-th and (*stringlength* + 2)-th columns contain the real x value, used as auxiliary information in order to check the algorithm's evolution, and the corresponding $f(x)$, which is assumed to be the fitness value of the string in this case. Then the initialisation process can be completed using the following code:

```
function [pop] = initialise(popsize, stringlength, fun);
    pop = round(rand(popsize, stringlength + 2));
    pop(:, stringlength + 1) = sum(2.^(size(pop(:,1:stringlength),2) - 1:-1:0).
        *pop(:,1:stringlength))*(b - a)/(2.^stringlength - 1) + a;
    pop(:, stringlength + 2) = fun(pop(:, stringlength + 1));
    end
```

In the above routine, we first generate the binary bits randomly, and then replace the (*stringlength* + 1)-th and (*stringlength* + 2)-th columns with real x values and objective function values, where *fun* is the objective function, usually denoted by a .m file.

3.2 *Crossover*

Crossover takes two individuals *parent*1, *parent*2, and produces two new individuals *child*1, *child*2. Let *pc* be the probability of crossover, then the crossover operator can be implemented as follows:

```
function [child1, child2] = crossover(parent1, parent2, pc);
  if (rand < pc)
    cpoint = round(rand*(stringlength - 2)) + 1;
    child1 = [parent1(:,1:cpoint) parent2(:,cpoint1 + 1:stringlength)];
    child2 = [parent2(:,1:cpoint) parent1(:,cpoint1 + 1:stringlength)];
    child1(:, stringlength + 1) = sum(2.^(size(child1(:,1:stringlength),2) - 1:-1:0).
        *child1(:,:stringlength))*(b - a)/(2.^stringlength - 1) + a;
    child2(:, stringlength + 1) = sum(2.^(size(child2(:,1:stringlength),2) - 1:-1:0).
        *child2(:,1:stringlength))*(b - a)/(2.^stringlength - 1) + a;
    child1(:, stringlength + 2) = fun(child1(:, stringlength + 1));
    child2(:, stringlength + 2) = fun(child2(:, stringlength + 1));
  else
    child1 = parent1;
    child2 = parent2;
  end
  end
```

At the top of the *crossover* routine, we determine whether we are gong to perform crossover on the current pair of parent chromosomes. Specifically, we generate a random number and compare it with the probability parameter *pc*. If the random number is less than *pc*, a crossover operation is performed; otherwise, no crossover is performed and the parent individuals are returned. If a crossover operation is called for, a crossing point *cpoint* is selected between 1 and *stringlength*. The crossing point *cpoint* is selected in the function *round*,

which returns a pseudorandom integer between specified lower and upper limits (between 1 and *stringlength* -1). Finally, the partial exchange of crossover is carried out and the real values and fitness of the new individuals *child*1, *child*2, are computed.

3.3 *Mutation*

Mutation alters one individual, *parent*, to produce a single new individual, *child*. Let *pm* be the probability of mutation, then as in the *crossover* routine, we first determine whether we are going to perform mutation on the current pair of parent chromosomes. If a mutation operation is called for, we select a mutating point *mpoint*, and then change a true to a false (1 to 0) or vice versa. The real value and fitness of the new individual *child* are then computed:

```
function [child] = mutation(parent, pm):
    if (rand < pm)
      mpoint = round(rand*(stringlength − 1)) + 1;
      child = parent;
      child[mpoint] = abs(parent[mpoint] − 1);
      child(:, stringlength + 1) = sum(2.^ (size(child(:,1:stringlength),2) − 1: − 1:0).
            *child(:,1:stringlength))*(b − a)/ (2.^stringlength − 1) + a;
      child(:, stringlength + 2) = fun(child(:, stringlength + 1));
    else
      child = parent;
    end
    end
```

3.4 *Selection*

The selection operator determines which of the individuals will survive and continue in the next generation. The selection operator implemented here is *roulette wheel* selection and this is perhaps the simplest way to implement selection. We first calculate the probabilities of each individual being selected, based on equation (1). Then the partial sum of the probabilities is accumulated in the vector *prob*. We also generate a vector *rns* containing normalised random numbers, by comparing the elements of the two vectors *rns* and *prob*, we decide the individuals which will take part in the new population:

```
function [newpop] = roulette(oldpop);
      totalfit = sum(oldpop(:,stringlength + 2));
      prob = oldpop(:,stringlength + 2) / totalfit;
      prob = cumsum(prob);
      rns = sort(rand(popsize,1));
      fitin = 1; newin = 1;
      while newin <= popsize
      if (rns(newin) < prob(fitin))
            newpop(newin,:) = oldpop(fitin,:);
            newin = newin + 1;
      else
            fitin = fitin + 1;
      end
      end
```

3.5 *Accessing on-line information*

We have shown how the three main pieces of CGA may be easily coded and easily understood. After implementing the key genetic operators, implementation of the main program is straightforward. At the moment, the World Wide Web and the Internet provide some quality MATLAB source codes for different kinds of genetic algorithms. Chris Houck, Jeff Joines, and Mike Kay at North Carolina State University have developed two versions of the genetic algorithm for function optimization using both MATLAB 4 and MATLAB 5. The WWW site is: http://www.ie.ncsu.edu/mirage. Andrew F. Potvin has also implemented a simple version of genetic algorithm which is available on the WWW site: http://unix.hensa.ac.uk/ftp/mirrors/matlab/contrib/v4/optim/genetic/.

It should be pointed out that although the above Toolboxes are very useful in solving many engineering problems, they are sometimes too comprehensive for the students. In many situations, the students need to modify the genetic operators to meet their own problem-dependent requirements. However, they found it very difficult to change some parts of the above GA Toolboxes, because the whole package is heavily interconnected. With the source codes provided in this section, the students found it quite easy to modify the genetic operators and invent some new genetic operators. This is helpful in promoting the students' creativity, which is necessary and important in the new teaching process.

4 AN ILLUSTRATIVE EXAMPLE

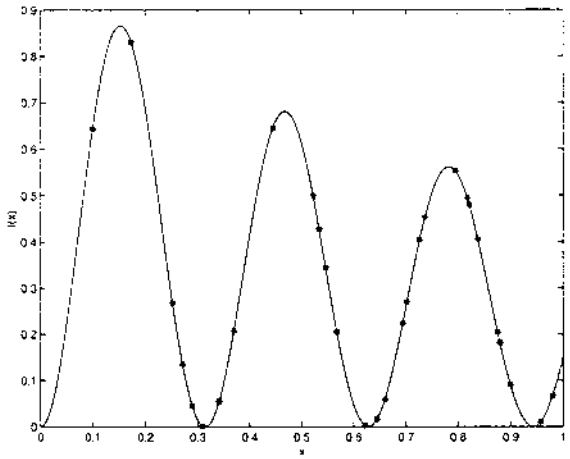
This section demonstrates that the CGA is able to find the global or near-global maximum of multi-modal functions. As an example, the following one-variable function is considered:

$$f(x) = \frac{\sin^2(10x)}{1+x} \tag{2}$$

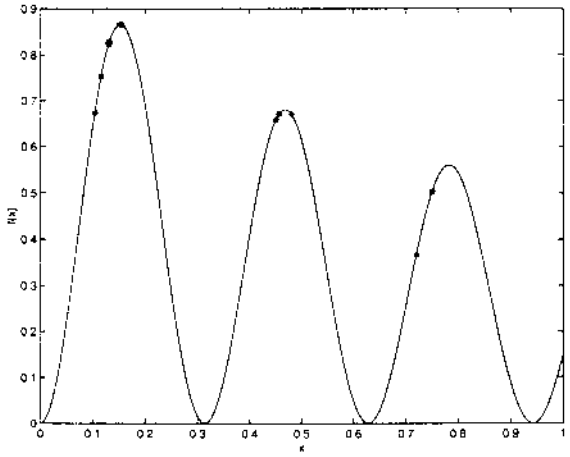
for x belonging to the domain $[0, 1]$. The characteristic of the function is plotted in Fig. 4(a). As shown in the figure, the function has three maxima, with a global maximum achieved at $x = 0.1527$.

Several trials have been performed changing the fundamental parameter values. Below, a relevant example, with $popsize = 30$, $stringlength = 20$, $pc = 0.95$ and $pm = 0.05$ is reported. Fig. 4(a) shows the initial population for the optimisation of a one-variable optimisation: the $*$ represent the individuals of the population, randomly generated in the function domain. Fig. 4(b) shows the population after 50 generations of the CGA, and Fig. 4(c) gives the population after 100 generations of the CGA.

As can be observed from Fig. 4(a), at the first step, the population (represented by $*$ in the figures) is randomly distributed in the whole function domain. During the evolution of the algorithm, Fig. 4(b), the individuals of the population tend to concentrate at the peaks of the function and at convergence, Fig. 4(c), most of the individuals are clustered near the desired maximum value of the function.



(a)

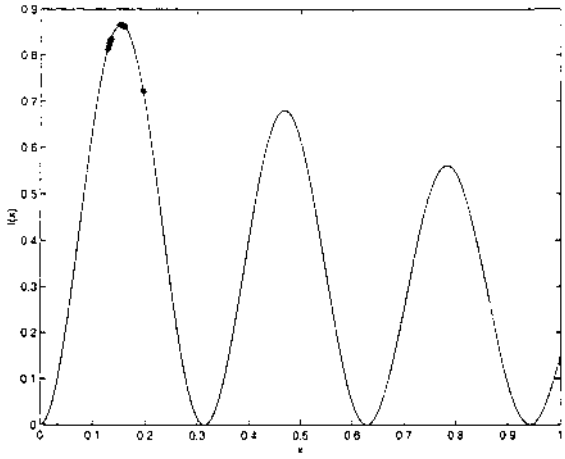


(b)

FIG. 4 Performance of the CGA with a multi-modal function.

5 AN APPLICATION EXAMPLE

In this section, we provide an application example of the developed GA in designing robust controller for uncertain control systems. Robust control under parametric uncertainty is an important research area with many practical applications. When the system has a general nonlinear uncertainty structure, the usual approach is to overbound it by an interval dynamical system. The



(c)

FIG. 4 (Continued.)

robust controller design can be transformed into solving an eigenvalue optimisation problem, which involves optimising the maximum real part of eigenvalues of the characteristic polynomial for the uncertain plant. This kind of problem is a typical non-smooth and nonconvex optimisation problem, and may have several local optima. We demonstrate that GA is capable of solving this kind of problem. A practical flywheel–shaft–flywheel system, including uncertainty in the length of the shaft and shaft damping, is employed to carry out the numerical simulation.

5.1 *Uncertain control systems*

Let us consider the following single-input single-output (SISO) linear systems:

$$a_n(\mathbf{q})\frac{\mathrm{d}^n y}{\mathrm{d}t^n} + a_{n-1}(\mathbf{q})\frac{\mathrm{d}^{n-1} y}{\mathrm{d}t^{n-1}} + \cdots + a_0(\mathbf{q})y$$

$$= b_m(\mathbf{q})\frac{\mathrm{d}^m u}{\mathrm{d}t^m} + b_{m-1}(\mathbf{q})\frac{\mathrm{d}^{m-1} u}{\mathrm{d}t^{m-1}} + \cdots + b_0(\mathbf{q})u \tag{3}$$

where n is the order of the model, y is the output variable, u is the input variable, and $\mathbf{q} = \{q_1, q_2, \ldots, q_k\}$ is the design parameter vector. Furthermore, we acknowledge uncertainties in the design parameters by noting the design parameter vector $\mathbf{q}$ lies in a box $Q = [q_1^-, q_1^+] \times [q_2^-, q_2^+] \times \cdots \times [q_k^-, q_k^+]$. The transfer function of (3) is defined by

$$G(s, \mathbf{q}) = \frac{Y(s, \mathbf{q})}{U(s, \mathbf{q})} = \frac{\sum_{i=0}^{m} b_i(s, \mathbf{q}) s^i}{\sum_{j=0}^{n} a_j(s, \mathbf{q}) s^j} = \frac{N(s, \mathbf{q})}{D(s, \mathbf{q})}. \tag{4}$$

For convenience, we define the entire family of uncertain plants as

$$\mathbf{G}(s) = \{G(s, \mathbf{q}) \mid \mathbf{q} \in Q\}. \tag{5}$$

We also consider a feedback controller, $F(s, \mathbf{p}) = F_1(s, \mathbf{p})/F_2(s, \mathbf{p})$, where the numerator and denominator of F are polynomials in s, and $\mathbf{p} = \{p_1, p_2, \ldots, p_l\}$ is the parameter vector lying in a box

$$P = [p_1^-, p_1^+] \times [p_2^-, p_2^+] \times \cdots \times [p_l^-, p_l^+].$$

The task of robust controller design considered in this section is to determine the optimal value of $\mathbf{p}$ so that the feedback control system is stable for all parameters $\mathbf{q} \in Q$.

For the feedback system shown in Fig. 5, the characteristic polynomial is:

$$H(s) = F_2(s, \mathbf{p})D(s, \mathbf{q}) + F_1(s, \mathbf{p})N(s, \mathbf{q}) \tag{6}$$

and this polynomial varies over the corresponding uncertainty set

$$\mathbf{H}(s, \mathbf{q}, \mathbf{p}) = \{F_2(s, \mathbf{p})D(s, \mathbf{q}) + F_1(s, \mathbf{p})N(s, \mathbf{q}) \mid \mathbf{q} \in Q, \mathbf{p} \in P\} \tag{7}$$

Using Fig. 5, we cite the transfer functions pertinent to control

$$T^y(s, \mathbf{q}, \mathbf{p}) = \frac{Y(s)}{R(s)} = \frac{F(s, \mathbf{p})G(s, \mathbf{q})}{1 + F(s, \mathbf{p})G(s, \mathbf{q})}. \tag{8}$$

The robust controller design can be considered as determining the parameter value $\mathbf{p} \in P$ so that the feedback control system is stable for all $\mathbf{q} \in Q$. However, the presence of the k-dimensional uncertainty vector, $\mathbf{q}$, complicates the analysis of the uncertain transfer functions such as (4) and equations such as (6). Thorough robust analysis requires that the behaviour of every point in the uncertainty space, i.e. all $\mathbf{q} \in Q$, must be checked. As this is impossible, a common technique involves creating a grid of points for each $q_i \in [q_i^-, q_i^+]$, $i = 1, \ldots, k$, and analysing behaviour over this k-dimensional grid. This approach is computationally intensive and inexact — one can never be certain that a finer grid might not change a robust stability prediction. In the next subsection, we discuss solving this problem using GA optimisation approach.

5.2 *Robust controller design via GA*

As mentioned above, the main task of robust controller design is to determine the optimal value of $\mathbf{p}$ so that the feedback control system is stable for all parameters $\mathbf{q} \in Q$. This can be transformed into solving eigenvalue optimisa-

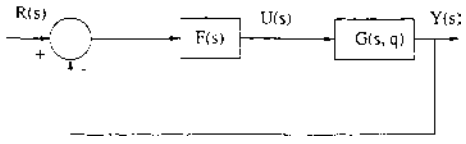


FIG. 5 Uncertain feedback control plant.

tion problems. To be more precise, the optimisation problem may be stated mathematically as

$$\min_{\mathbf{p} \in P} g(\mathbf{p}) \qquad \text{and} \qquad g(\mathbf{p}) = \max_{i, \mathbf{q} \in Q} \mathrm{Re}(\lambda_i(H(s, \mathbf{q}, \mathbf{p}))). \tag{9}$$

This is a typical min–max optimisation problem, and the following result is straightforward[4].

Proposition If (p^*, q^*) is the global optimum of the problem (9), and the corresponding objective value is less than 0, then the uncertain system (3) with feedback controller $F(p^*, s)$ is stable for all $\mathbf{q} \in Q$.

This proposition implies that, to design the robust controller for uncertain system (3), an alternative approach is to develop a reliable approach to find the global optima of problem (9). Recently, GA has been considered as viable and promising techniques for global optimisation of complex functions. When applying GA to the min–max problem (9), the fitness scaling technique in Ref. [3] and the nesting technique in Ref. [5] are always used. The former technique is to keep the fitness positive and avoid large deviation of fitness. The latter technique involves an inner loop to maximise the $\mathrm{Re}(\lambda_i(H(s, \mathbf{q}, \mathbf{p})))$ for fixed parameters $\mathbf{p}$, and an outer loop to minimise $g(\mathbf{p})$ over the set P.

Let us consider the practical shaft–flywheel system illustrated in Fig. 6. The input to the system is a torque applied to the first flywheel, and the output is the angular velocity of the second flywheel. We model the shaft–flywheel system as two torsional inertias separated by a torsional spring and a torsional dashpot. The transfer function for this system is

$$G(s, \mathbf{q}) = \frac{N(s, \mathbf{q})}{D(s, \mathbf{q})} = \frac{\dfrac{C_\mathrm{D}}{J_1 J_2} s + \dfrac{K}{J_1 J_2}}{s^3 + C_\mathrm{D}\left(\dfrac{1}{J_1} + \dfrac{1}{J_2}\right) s^2 + K\left(\dfrac{1}{J_1} + \dfrac{1}{J_2}\right) s} \tag{10}$$

where J_1 and J_2 are the inertia of the left flywheel and one-half of the shaft inertia and the inertia of the right flywheel and one-half of the shaft inertia, respectively. K is the torsional stiffness of the shaft, and C_D is the torsional

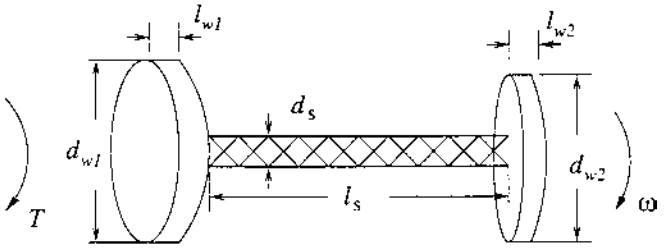


FIG. 6 Flywheel–shaft–flywheel plant.

damping of the shaft. We use standard relations to compute the torsional inertias and stiffness, i.e.

$$J = \frac{\rho \pi l d^4}{32} \tag{11}$$

$$K = \frac{G \pi d^4}{32 l} \tag{12}$$

where ρ is material density, l is the length of a circular shaft, d is the diameter, and G is the shear modulus. Equation (11) also applies to the flywheels. We assume proportional damping for the shaft, where $C_D = C_0 K$.

We consider the case in which the uncertain design parameter vector, $\mathbf{q}$, consists of C_0, the proportional damping coefficient, the l_s, the shaft length. This vector lies in the box

$$Q = \{\mathbf{q} \mid C_0 \in [0.0008, 0.0012], \qquad l_s \in [0.9, 1.1]\}$$

and the other parameters have the values: $d_{w1} = 0.20$, $l_{w1} = 0.02$, $d_s = 0.01$, $d_{w2} = 0.15$, and $l_{w2} = 0.02$.

We assume the controller to be designed having the following structure:

$$F(s, \mathbf{p}) = 0.234 \frac{(s + p_3)(s^2 + p_4 s + p_5)}{s(s^2 + p_1 s + p_2)} \tag{13}$$

The developed GA is employed to solve the optimisation problem (9), involving the transfer function $G(s, \mathbf{q})$ in (10) and feedback controller $F(s, \mathbf{p})$ in (13). The population size and chromosome length in GA are chosen as 120 and 100, respectively. The probabilities of crossover and mutation in GA are chosen as

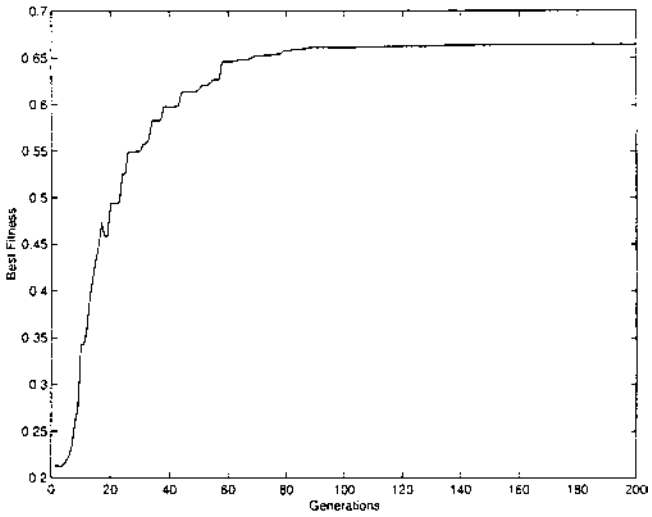


FIG. 7 Optimisation procedure for a robust controller with GA.

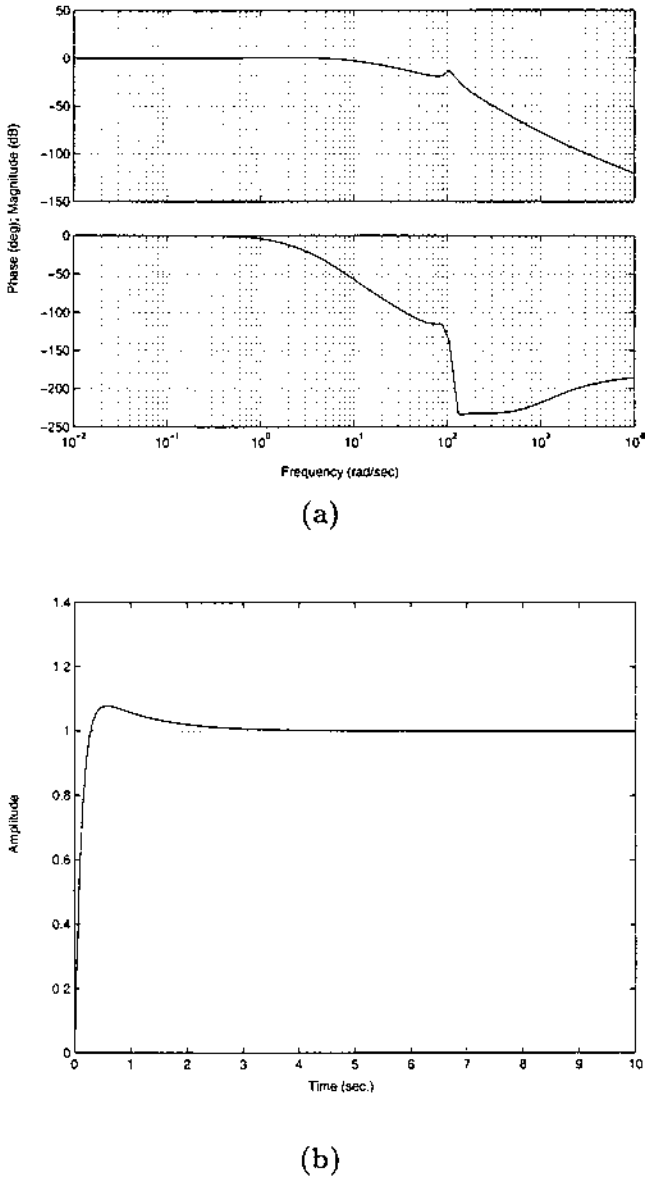
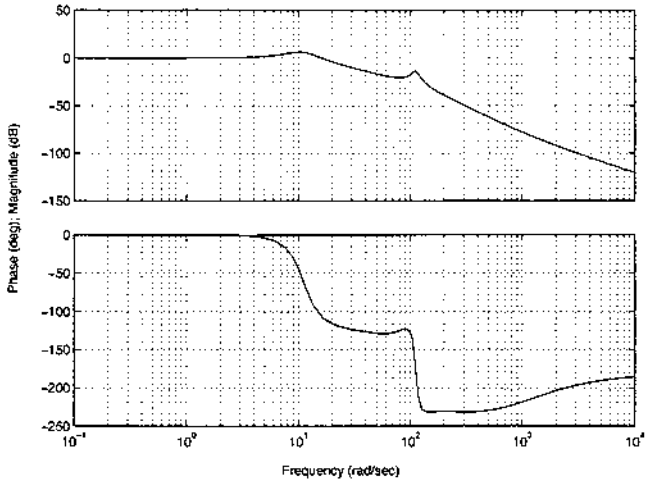
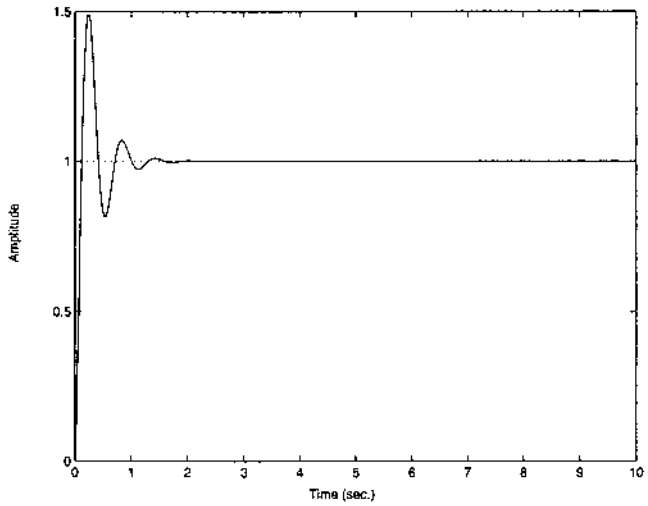


FIG. 8 Performance of an uncertain system with a robust controller.

0.95 and 0.05. Using the fitness scaling technique in Ref. [3] and the nesting technique in Ref. [5], the optimal solution obtained using GA is: $p_1 = 212.60$, $p_2 = 11024.80$, $p_3 = 15.12$, $p_4 = 79.86$, $p_5 = 11026.84$. The optimisation procedure is shown in Fig. 7, where the best fitness in each population with respect to generations is plotted. Fig. 8 and Fig. 9 show the Bode plots and the step



(a)



(b)

FIG. 9 Performance of an uncertain system with a robust controller.

responses of the feedback control system, when the uncertain parameters take $C_0 = 0.0008$, $l_s = 1.0$ and $C_0 = 0.0012$, $l_s = 1.1$, respectively. Furthermore, the uncertain plant (10) with the feedback controller (13) can be guaranteed stable by Kharitonov's theorem.

6 CONCLUSION

In this paper, an attractive approach for teaching genetic algorithm has been presented. This approach is based primarily on using MATLAB in implementing the genetic operators: crossover, mutation, and selection. An advantage of using such an approach is that the student becomes familiar with some advanced features of MATLAB, and furthermore, with the availability of other MATLAB Toolboxes such as The Control Systems Toolbox, Neural Network Toolbox and Fuzzy Logic Toolbox, it is possible for the student to develop genetic algorithm-based approaches to designing intelligent systems, which could lead to his/her final year or MSc project.

REFERENCES

[1] *MATLAB 5*, The Math Works Inc., Natick, MA (1997)
[2] Holland, J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor (1975)
[3] Goldberg, D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley (1989)
[4] Borie, J. A., *Modern Control Systems*, Prentice-Hall International (1986)
[5] Sebald, A. V. and Schlenzig, J., 'Minimax design of neural net controller for highly uncertain plants', *IEEE Trans. Neural Networks*, **5**, pp. 73–82 (1994)

ABSTRACTS – FRENCH, GERMAN, SPANISH

Enseignement d'algorithme génétique par MATLAB

Dans cet article une approche attractive pour enseigner l'algorithme génétique (GA) est présentée. Cette approche est basée sur l'utilisation de MATLAB pour implémenter les opérateurs génétiques: mutation, sélection et croisement. Un exemple illustratif détaillé est présenté pour démontrer que GA est capable de trouver un optimum global ou quasi-global pour des fonctions multi-modales. Une application de GA dans la conception d'un contrôleur robuste pour des systèmes à contrôle incertain est également donnée pour montrer son potentiel dans la conception de systèmes intelligents en ingéniérie.

Genetische Algorithmen mit MATLAB lehren

In diesem Bericht wird ein attraktives Vorgehen zum Lehren genetischer Algorithmen (GA) vorgestellt. Dieses Vorgehen beruht in erster Linie auf der Verwendung von MATLAB in der Durchführung der genetischen Operatoren: Überkreuzung, Mutation und Selektion. Ein ausführliches illustratives Beispiel wird vorgestellt, um zu beweisen, daß GA fähig ist, globale oder fast globale optimale Lösungen multimodaler Funktionen zu finden. Eine Anwendung von GA bei der Auslegung eines robusten Reglers für ungewisse Steuersysteme wird ebenfalls angegeben, um dessen Potential in der Entwicklungstechnik intelligenter Systeme zu zeigen.

Enseñanza de algoritmos genéticos empleando MATLAB

En este artículo se presenta una aproximación atractiva para la enseñanza de algoritmos genéticos (AG). Esta aproximación se basa, en primer lugar, en emplear MATLAB para la implementación de los operadores genéticos: cruzamiento, mutación y selección. Se presenta un ejemplo ilustrativo detallado para demostrar que el AG es capaz de buscar soluciones globales o casi globales óptimas de funciones multimodales. Se muestra así mismo una aplicación de AG en el diseño de controladores robustos para sistemas de control con incertidumbres que muestra su potencial en el diseño de sistemas de ingeniería inteligentes.