GENETIC ALGORITHM OPTIMISATION SOFTWARE IN FORTRAN

by

A Anderson

A R Simpson

Research Report No. R136
March 1996
GENETIC ALGORITHM OPTIMISATION SOFTWARE
IN FORTRAN

by

A Anderson¹
A R Simpson²

Department of Civil and Environmental Engineering
The University of Adelaide

Research Report No. R136
March 1996

¹. Department of Mechanical Materials and Manufacturing Engineering, University of Newcastle upon Tyne, United Kingdom
². Department of Civil and Environmental Engineering, University of Adelaide, Australia
GENETIC ALGORITHM OPTIMISATION SOFTWARE IN FORTRAN

ABSTRACT

Existing shareware for Genetic Algorithm optimisation tends to be in the programming languages of Pascal and C. The Fortran language is in many ways less suitable for this application, yet remains widely used in the engineering community, and the availability of a Fortran code may encourage wider use of Genetic Algorithms. The FORTRAN GA implementation is a ‘simple’ genetic algorithm based on real number genes, with generational reproduction using tournament selection with complete replacement for chromosomes, followed by the operations of cross-over (without elitism), mutation and inversion at specific probabilities. A number of innovative features have been included in the genetic algorithm implementation. It is designed to allow investigation of variable population size, tournament selection with different numbers of competitors and victors, cross-over with different numbers of cross-over points and participants, and the possibility of refreshing the population with incomers every few generations. As the implementation is intended to encourage the investigation of real engineering problems, which often have complex simulation demands, it allows duplicate chromosomes to be removed at each generation (to avoid redundant simulation) and also allows automatic replacement of physically infeasible chromosomes (to avoid simulation failures). In practical engineering applications with multiple local optima, patterns of close to optimal performance may be of as much interest as a single optimum point, so provision is made to automatically record a specified number of best chromosomes found throughout all generations. Full documentation is provided to support further adaptations.

ACKNOWLEDGMENTS

This work was carried out when one of the authors was a Visiting Scholar in (and using facilities provided by) the Department of Civil and Environmental Engineering at the University of Adelaide, while on sabbatical leave granted by the University of Newcastle-upon-Tyne with financial support from the British Council ALIS programme.
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Why?</td>
<td>1</td>
</tr>
<tr>
<td>1.2 What?</td>
<td>1</td>
</tr>
<tr>
<td>2. Description of Fortran Implementation</td>
<td>4</td>
</tr>
<tr>
<td>2.1 General Programming Principles</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Genes, Chromosomes and Parent/child Populations</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Fitness and Integer Pointer Vectors</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Integer Pointer Vectors for Reproduction Operations</td>
<td>8</td>
</tr>
<tr>
<td>2.5 Input/output, Process Recording and Results</td>
<td>8</td>
</tr>
<tr>
<td>3. Outline of GA Selection and Reproduction Processes</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Generational Reproduction with Replacement</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Tournament Selection</td>
<td>13</td>
</tr>
<tr>
<td>3.3 Cross-over</td>
<td>14</td>
</tr>
<tr>
<td>3.4 Mutation and Inversion</td>
<td>16</td>
</tr>
<tr>
<td>4. GA Operation Options</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Selection and Reproduction Operations</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Chromosome Size</td>
<td>20</td>
</tr>
<tr>
<td>4.3 Population Size</td>
<td>20</td>
</tr>
<tr>
<td>4.4 Deliberate Changes to Population Members</td>
<td>22</td>
</tr>
<tr>
<td>4.5 Record of Best Chromosomes</td>
<td>23</td>
</tr>
<tr>
<td>4.6 Number of Generations and Convergence Criteria</td>
<td>23</td>
</tr>
<tr>
<td>4.7 Random Number Seed</td>
<td>24</td>
</tr>
<tr>
<td>5. An Example Application</td>
<td>24</td>
</tr>
<tr>
<td>5.1 What is Necessary to Use this Code for a Specific Application</td>
<td>24</td>
</tr>
<tr>
<td>5.2 The User-Defined Calling Main Program</td>
<td>25</td>
</tr>
<tr>
<td>5.3 The Simulation Code Interface at Subroutine SIMULA</td>
<td>25</td>
</tr>
<tr>
<td>5.4 Case-Specific Subroutines NEWCHAR and MUTGEN</td>
<td>26</td>
</tr>
</tbody>
</table>
5.5 Summary of Option and Case Declarations in Subroutine GENALG

5.5.1 Selection of GA options (Section 4)
5.5.2 Memory Requirements

5.6 Description of Appendix C Output for the $x^2$ Example

6. Availability of and Responsibility for Using the Code

7. Conclusions

8. References

Appendix A - Code Listing

Appendix B - Module Descriptions and Flow Charts
B.1 Principal GA subroutine GENALG
B.2 User-defined interface subroutines SIMULA, NEWCHR and MUTGEN
B.3 Eliminating duplicates in the populations - subroutines INIPOP and UNIQUE
B.4 Simulations with population management - subroutines REFRSH and RNSIMR
B.5 Fitness statistics (subroutine POSTAT) and best chromosomes from all generations (subroutine CREAM)
B.6 Population size (function NEWPOP) and reproduction pool selection (subroutine TOURN)
B.7 Reproduction - cross-over (subroutine XOVERT)
B.8 Reproduction - mutation and inversion (subroutines MUTATE, MUTCHR, INVERT, INVCHR)
B.9 Recording and outputting results (subroutines POPOUT and RESULT)
B.10 Pointer integer vectors - subroutines PKPART, MKLIST, MKMARK and MKSQL
B.11 Standard numerical routines - functions CHSAME and RANDOM, subroutines ISORTI and IRANKP

Appendix C - Goldberg’s $x^2$ Example Output
1. INTRODUCTION

Over the past five years a focus of research at the University of Adelaide has been the development of a methodology for applying Genetic Algorithm optimisation to water distribution pipe network design. A number of practical networks have been considered, eg Murphy et al (1993), Murphy et al (1994) and Simpson et al (1995; 1996). Most previous Genetic Algorithm implementations by the research group at the University of Adelaide have been in Pascal.

1.1 Why?

Why write a Genetic Algorithm (GA) implementation code in the FORTRAN programming language, especially as this lacks the data-typing features which make other languages such as Pascal and C more suitable for this application?

Many engineers remain more familiar with Fortran and much simulation software, on which GAs are dependent, remains solely available in that language. While there is no practical barrier to combining these with Pascal or C GA routines, nevertheless the confidence of many engineers in using Fortran may result in a greater willingness to investigate GAs for optimisation if Fortran GA routines are available.

Why write an entirely new GA code instead of simply adapting or translating one of the existing public domain codes (eg Goldberg 1989, Davis 1991, Smith et al 1994)?

Firstly, the lack of certain data-typing features in Fortran (compared with Pascal or C) means that some aspects of implementation must necessarily be different. Secondly, and more importantly, experience with GAs at the University of Adelaide (Murphy et al 1993a, Simpson and Goldberg 1994) indicates that many aspects of GA implementation continue to require investigation and suitable code must be prepared to accommodate some of these (as described below). Thirdly, availability of fully-documented code in a widely-known programming language may facilitate further adaptations by future investigators.

1.2 What?

What kind of GA implementation has been coded?

The basic algorithm is a “simple” GA (Goldberg 1989) using chromosomes made up from real number (0.0 to 1.0) genes and generational, rather than steady-state, reproduction with tournament selection. Davis (1991) promotes
steady-state reproduction for efficiency but concedes that generational reproduction is probably more robust, robustness being a key feature of GA optimisation (Goldberg 1989). Simpson and Goldberg (1994) found that generational reproduction with tournament selection outperformed a version of steady-state reproduction.

While there are arguments in favour of genes being made up from the smallest possible components (ie binary bits as alleles), eg Davidor (1991), experience at the University of Adelaide has been that tournament selection and an adequate population size are the most significant features of the GA implementation and that with this there appears to be no disadvantage in using compact real-number genes in real practical problems (Davis 1991), as in Evolutionary Strategy (Aesche and Simpson 1994). Certainly, for many real engineering problems with parameters or variables that are not discrete, real-number genes may be more convenient and their use reduces machine or system-dependency of Fortran code.

What are the main features of GA implementation that the new code allows to be investigated?

Firstly, for practical engineering problems, such as pipe network optimisation, previous work at the University of Adelaide (Simpson and Goldberg 1994), including student projects (Brumfield et al 1991), has drawn attention to the selection of an appropriate population size. While Davis (1991) has argued in favour of reproduction operation (cross-over, mutation, inversion) probabilities which adapt to the population characteristics and Davidor (1991) has discussed adaptive chromosome length, the same arguments (with some biological analogy justification) can be applied to an adaptable variable population size. Alternatively, it has also been suggested that problems with small populations leading to premature convergence can be addressed by allowing the population to be periodically refreshed by incomers to ensure diversity (a form of immigration). Both concepts have therefore been implemented in the Fortran GA code developed here.

Secondly, while investigations at the University of Adelaide have supported the preference for tournament over weighted probability ("roulette-wheel") selection (with no fitness scaling), to date this has been mainly for binary (two competitor) tournaments. Other tournament sizes have also been tried (Simpson and Goldberg 1994). The new Fortran GA implementation allows any number of competitors (Smith et al 1994) and victors in each tournament round.

Thirdly, Simpson and Goldberg (1994) have established that cross-over is an important operation, but as yet there has been little investigation of different forms of this compared with mutation. Therefore, the Fortran GA
implementation allows cross-over at any number of points (from 1 to one less than L, the number of genes) in the chromosome, including the commonly adopted values of one or two cross-over points. Cross-over at (L-1) genes is equivalent to uniform cross-over. In addition, cross-over is usually between pairs of chromosomes, but the implementation allows rotation of the selected genes between any number (between two and population size) of chromosomes. While experience at the University of Adelaide and by Goldberg at the University of Illinois has been that mutation is relatively insignificant in the effectiveness of the GA search, both this and the operation of inversion (Davis 1991) have been incorporated. Inversion is easily defined (although omitted in, eg, Smith et al 1994) but there are many variants of the mutation operation in the literature and implementation is likely to be strongly case-specific in practical engineering problems, eg the use of adjacency or creeping mutation to move to the next discrete pipe size in pipe network optimisation (Murphy et al 1993b). The Fortran GA implementation recognises this and allows for user-defined gene mutation.

What other features have been incorporated into the new Fortran GA code that may not be generally available in other GA implementations?

Firstly, Davis (1991) points out that in many practical applications the simulation time may significantly exceed any overhead arising from the GA code and recommends that populations be scanned to remove duplicate chromosomes and avoid redundant simulation. As the presence of duplicates is a particular risk with tournament selection (with accompanying loss of population diversity), this has been implemented in the Fortran GA.

Secondly, in practical problems there is often the related problem of how to handle physically inappropriate or infeasible chromosomes that may arise from the essentially random application of the GA reproduction operations. Practice at the University of Adelaide has been to incorporate penalty functions into the fitness evaluation following simulation (Murphy and Simpson 1993, Simpson et al 1994) but the problems of identifying appropriate and effective penalties have been noted (Brumfield et al 1991, Dandy et al 1993, Simpson et al 1994) and there may be situations in which physically infeasible chromosomes could cause undesirable failure of the simulation (Hassanli and Dandy 1994). Consequently, the option of replacing infeasible members of a population at each generation has been also implemented in the Fortran GA.

Finally, what features mentioned in the research literature have not been implemented in the Fortran GA code?

These are, of course, almost too numerous to mention, but five in particular seem worth noting. Firstly, an advantage of tournament as opposed to weighted probability ("roulette wheel") selection in that there is no need for fitness
scaling (which could, in any case, be case-specific implemented at the simulation coding interface) and this is not implemented.

Secondly, in tournament selection with groups of more than two competitors (as allowed for in this implementation), Simpson and Goldberg (1994) suggest the use of weighted-probability ("roulette wheel") selection within these groups, but only a simple fitness comparison has been implemented.

Thirdly, while adaptable variable population size has been implemented, adaptable variable reproduction operator probabilities (Davis 1991) have not been implemented. However, this could easily be added and there is a comment to this effect at the appropriate point in the code.

Fourthly, in some implementations (eg Davis 1991 or with Evolution Strategy as in Aesche and Simpson 1994) the reproduction operations (cross-over, mutation and, if implemented, inversion) are made mutually exclusive (ie mutation may be applied only to chromosomes which have not been crossed, etc) but this is not the case in this Fortran GA implementation, in which the three operations are applied separately to the population as a whole (though for each separate operation, each chromosome may be involved once only).

Finally, at each generation, reproduction is by complete replacement with no "elitism" or "preselection" (ie best of previous generation would proceed to next generation). It is a feature of tournament selection that the fittest in the parent population must be selected and the least fit must be discarded, but the subsequent reproduction operations (cross-over, mutation, inversion) in this implementation may then alter the former and there is no guarantee that they proceed into the next generation. In the terminology of Evolution Strategy (Aesche and Simpson 1994), this is a $\mu$+$\lambda$ rather than $\mu$+$\lambda$ implementation.

2 DESCRIPTION OF FORTRAN IMPLEMENTATION

2.1 General Programming Principles

The source code is listed in Appendix A. Module descriptions and, where appropriate, flow charts are given in Appendix B. It is hoped that the code is ANSI standard FORTRAN-77 throughout with no hardware or operating system dependent features. To facilitate its comprehensibility:

A. All variable names and function types are declared throughout (implied typing could be suppressed at compilation).

B. Structured code is used throughout with no jumps (GOTO) or statement labels except where incorporated into ANSI standard
Fortran-77 constructions for “repeat-until” or “do-while” loops or required by standard DO-CONTINUE loops.

C. Modular code is used throughout with all data passed to sub-programs by parameters, avoiding use of COMMON statements (except to hold random number data in block /RDM/ for function RANDOM and variable population parameters in block /POSIZE/ for function NEWPOP), avoiding “side-effect” changes to global variables and initialising all variables locally.

D. PARAMETER declarations are used only for array bounds with DATA declarations for all other values.

E. As presented, the FORTRAN GA code could be made more efficient by using temporary storage for terms (mostly array indices) which are evaluated more than once, but these repeat evaluations are left in to aid readability.

F. All modules have been tested separately and independently of high-level code (but see disclaimer in Section 6).

An essential feature of the GA optimisation process is that it is independent of the application being optimised. Therefore the implementation presented here is in the form of a subroutine GENALG which contains all the necessary operational/option information within its declarations. To improve flexibility in certain applications these declarations could be replaced by parameters to the subroutine or COMMON declarations, but in Fortran this may have implications for dynamic array declarations where the variables affect these (ie MAXGEN, MAXPOP, NGENE and NCREAM). With these declared in subroutine GENALG, as far as is possible all array sizing in sub-modules is dynamic with arrays passed as parameters. However, in a few isolated instances further arrays need to be declared and in Fortran these must be declared locally (LIST in subroutines TOURN and XOVER, JCROSS in subroutine XOVER, JSTACK in subroutine IRANKP and possibly case-specific variables in subroutine SIMULA). It is essential that the sizes declared in these subroutines exceed those likely to occur (in Fortran it is perfectly acceptable to use less of the array at any subroutine call, but exceeding the declared bound will almost certainly corrupt other variable storage).

2.2 Genes, Chromosomes and Parent/child Populations

The basic building blocks of any GA implementation are the representations adopted for the genes, chromosomes and successive populations. The limitations of Fortran data types lead to the following:
(i) Genes:

Genes are simply real numbers in the range 0.0 to 1.0 (there is no checking in the code for this - it relies on appropriate values being assigned by function RANDOM and by user-code in subroutines SIMULA, NEWCHR and MUTGEN). As such there are no allele (components of genes). By restricting real number genes to values of 0.0 and 1.0 only, binary bits can be simulated (in which case strictly these become allele for groups which form genes, but the code simply refers to them as genes) as in Section 5 below. Genes are referenced as individual real numbers only in subroutine MUTGEN, otherwise as follows.

(ii) Chromosomes:

Chromosomes are vectors (one-dimensional arrays) of genes, of fixed length (ie number of genes) NGENE (declared as PARAMETER in subroutine GENALG). Genes are referenced as specifically indexed (from 1 to NGENE) members of these vectors. Gene position is irrelevant but is fixed (no provision is made for adaptive gene ordering as in “messy” Genetic Algorithms eg Goldberg 1989, Davidor 1991). Chromosomes are referred to as individual vectors only in subroutines MUTCHR, INVCHR, CHSAME and NEWCHR, otherwise as follows.

(iii) Populations:

Populations are two-dimensional real arrays of genes, with chromosomes forming columns of these arrays, and all the columns (chromosomes) forming the entire population. The population size (usually NPAR for parent and NCHILD for child in subroutine GENALG or NPOP elsewhere) is variable between the bounds MAXPOP and MINPOP (declared as PARAMETER and DATA, respectively, in subroutine GENALG), starting at NPAR which is the sole parameter to subroutine GENALG (if the fixed population option is chosen it simply remains at this value). Chromosome position is irrelevant but is fixed in each generation. Genes are referenced as components of the population array, with the second index identifying the chromosome (or member) column and the first index identifying the gene within that chromosome column. Chromosomes are referenced as vectors (ie columns of the population array) by setting the first index to 1 (head of column) and using the second index to choose the individual chromosome (from 1 to NPOP, NPAR or NCHILD). This ordering of indices
(gene first, chromosome second) is imposed by the way Fortran stores arrays if dynamic array sizing is to be used or if individual chromosome vectors are to be presented as subprogram parameters.

(iv) Successive populations:

In subroutine GENALG, simulation is carried out on a parent population PARENT(NGENE, NPAR) with NPAR members (or chromosomes), then selection (function NEWPOP and subroutine TOURN) produces a complete and separate child population CHILD(NGENE, NCHILD) with NCHILD members, which is then subjected to the reproduction operators (subroutines XOVER, MUTATE, INVERT and UNIQUE). Finally NPAR and array PARENT are overwritten by NCHILD and array CHILD, the former being lost (except for details recorded in subroutine GENALG) and the latter becoming the next generation. At any generation, only these two full populations are available (in arrays PARENT and CHILD).

2.3 Fitnesses and Integer Pointer Vectors

The driving force behind GA optimisation is the set of fitnesses, one for each chromosome generated by the simulation.

Fitnesses are stored as a vector (one-dimensional array) of real numbers (with no implicit restriction on value range) with the same array length and exactly corresponding one-to-one index order as the chromosomes in the population (second index of PARENT and CHILD arrays). In subroutine GENALG (and also in subroutine TOURN) the fitnesses are available only for the parent (PARENT) population in PARFIT(npar) and are established by the simulation in user-defined subroutine SIMULA. The code contains no fitness scaling (not necessary with tournament selection) but this could be user-defined in subroutine SIMULA if desired. In the lower level subroutines RNNSMR, CREAM, POSTAT and POPOUT, fitnesses for the population POP(npop) are in POPFIT(npop).

To avoid the computational expense of shuffling populations around, fitnesses are not rank ordered. Fitnesses correspond simply with their chromosome index and there is no significance in the order of the latter. To identify ranked fitnesses, integer vectors are used as pointers. These are set up from fitness vectors by subroutine IRANKP. Pointer vectors are indexed from 1 (least fit) to NPOP, etc (most fit) and contain the integer which is the index of the corresponding fitness and chromosome in the population. In subroutine GENALG, array LPRANK(maxpop) is the rank pointer for array PARENT(ngene, npar) with the fitness array PARFIT(maxpop). In subroutine
RNSIMR, array LFITRK(npop) is the rank pointer for array POP(ngene, npop) with fitness array POPFIT(npop). In subroutine POPOUT, array LPOPRA(npop) is the rank pointer for array POP(ngene, npop). In subroutine POSTAT, array LPRANK(npop) is the rank pointer for fitnesses array POPFIT(npop). As only the parent population has fitnesses, no pointers are needed for the child population.

2.4 Integer Pointer Vectors for Reproduction Operations

Integer pointer vectors are also used to implement reproduction operations. In subroutines TOURN and XOVER, the pointers in array LIST (maxpop) identify groups of chromosomes for tournament rounds or cross-over, respectively, by:

- referencing chromosomes within the population (of size <= maxpop),
- with values from 1 to maxpop, initially randomly ordered,
- of which, at any stage, the first NLEFT have not yet been processed (initially NLEFT = 0) whereas the remainder have been processed,
- and groups (usually pairs for crossover subroutine XOVER) are randomly and exclusively selected from the first NLEFT.

In subroutine XOVER the pointer array JCROSS(mxgene) identifies a sequence of genes within chromosomes for cross-over, by:

- referencing genes within chromosomes (of length <= mxgene),
- with the first NPOINT genes randomly and exclusively chosen from the range 1 to mxgene,
- and then rank ordered (using subroutine ISORTI).

Subroutines PKPART, MKLIST, MKMARK, MKSEQL and ISORTI along with function RANDOM create these integer pointer vectors.

2.5 Input/output, Process Recording and Results

Data input/output will be largely related to the case-specific simulation:

A. The software is written as subroutine GENALG to be called from a user-coded main program which will specify the bulk of data input and output.

B. The interface between the GA software and the user-coded application which will involve data transfers should be restricted to the following subroutines:
   - SIMULA - decode chromosomes, run simulation, calculate fitnesses;
- NEWCHR - generate a new chromosome (including those in the initial seed population); and

- MUTGEN - mutate a gene.

C. The remainder of the GA software should be independent of the application, just as the GA optimisation approach is independent of the application (Goldberg 1989).

However, the GA software provides for certain aspects of initiating and recording the GA optimisation process, some of which a user may choose to suppress, modify or expand. These features are restricted to four subroutines:

D. INIPOP does three things:

(i) It randomly generates an initial seed population. Function RANDOM uses the Knuth method (Press et al 1992) so given the same seed (variable SEED defined in a DATA declaration in function RANDOM) it will always produce the same sequence of random numbers and thus for a given problem the same initial population can be used to evaluate different options. However, a user could comment these lines out if an externally supplied initial population was required.

(ii) It prints out this seed population. This can be simply commented out or rewritten (e.g. to direct output to a file).

(iii) It also initialises the CREAM fitnesses (to be discussed below) - unless the call to subroutine CREAM is commented out this must be retained.

E. GENALG prints out some of the GA option settings to identify which version is being run. This could be commented out.

F. POPOUT is called every generation and simply prints the current parent population before it is overwritten by the new child population and thus lost. The call to this in subroutine GENALG would probably normally be commented out (or possibly it could be rewritten).

G. RESULT is called only when the GA is terminated and outputs data recorded throughout the process (described below). Again this could be commented-out or rewritten.

This last subroutine RESULT makes use of built-in process-recording which goes on throughout the GA cycle. This process-recording falls into two categories:
(a) Summary of each generation:

In subroutine GENALG, at each generation IGEN (starting at 1 up to a maximum maxgen), selected information is stored in arrays until it can be output at the GA termination:

- real two-dimensional array CEVOL(ngene, maxgen) stores in its columns the best (fittest) chromosome of each succeeding generation;

- real two-dimensional array REVOL(5, maxgen) stores in its columns five parent population fitness statistics (calculated in subroutine POSTAT) for each succeeding generation, ie 1 = maximum fitness, 2 = average fitness, 3 = median fitness, 4 = standard deviation, 5 = mean absolute deviation from median (these may also be used for the GA convergence termination criteria and in function NEWPOP or any additional code to give adaptive probabilities for current and past generation fitness characteristics); and

- integer two-dimensional array JEVOL(7, maxgen) stores in its columns seven GA operation statistics (calculated individually in the GA operation subroutines) for each succeeding generation, ie 1 = population size, 2 = number of population chromosomes refreshed or replaced if infeasible where either of these options used, 3 = number of chromosomes crossed-over, 4 = number of mutations, 5 = number of inversions, 6 = number at this generation added to CREAM (see below), 7 = total number of simulations (these may also be used for the GA convergence termination criteria - see Section 4.6 below).

(b) Best chromosomes found throughout entire process:

In many practical applications it may be useful to locate not only the absolute peak fitness but also any other peaks. To this end the variable NCREAM (declared as PARAMETER in subroutine GENALG) different best chromosomes from throughout the entire process are stored by subroutine CREAM until they can be output at the GA termination:

- array CCREAM(ngene, ncream) contains in its columns the best chromosomes, in no particular order, without duplicates;

- array FCREAM(ncream) contains fitnesses corresponding one-for-one with the same (second) index chromosomes in CCREAM; and

- array LCREAM(ncream) is an integer pointer vector containing the rank (1 = low, ncream = high) of the fitnesses and chromosomes.

These arrays (including LCREAM) must be initialised in subroutine INIPOP.
Finally, where there are locally-declared non-dynamic array bounds (see Sections 2.1 and 2.4), subroutines TOURN, XOVER and IRANKP will print warnings when these may be violated.

3. OUTLINE OF GA SELECTION AND REPRODUCTION PROCESSES

3.1 Generational Reproduction with Replacement

There is already significant (but probably healthy, by biological analogy) diversity in existing implementations of GAs and one of the present objectives is to add to this diversity. It is important, therefore, to define clearly the exact nature of the present GA implementation. The genes, chromosomes, populations and fitnesses are described in Sections 2.2 to 2.3 above. The basic form of generational reproduction with replacement implemented here proceeds as follows:

A. A starting (first generation, counter igen = 1) population size NPAR is defined by the user (parameter passed to subroutine GENALG) and an initial “seed” parent population array PAR(ngene, npar) is randomly generated by subroutine INIPOP, with each new chromosome checked to see that it does not duplicate an existing one (rejected if it does). User-defined code (in subroutine NEWCHR) allows the user to influence the form of the chromosomes generated for case-specific applications.

B. The FORTRAN GA then proceeds generation by generation until it is terminated (the conditions for this will be described in Section 4.6). At each generation the process is as follows:

(i) The parent population array PARENT(ngene, npar) is assigned corresponding fitnesses in array PARFIT(npar) through subroutines REFRESH and RNSIMR, the latter of which calls user-defined subroutine SIMULA to interface with the case-specific decoding/coding and simulation model. Population management may be applied at this stage (as described in more detail in Section 4.4 below).

(ii) Population statistics are determined (subroutine POSTAT) and details of the population are recorded (subroutine CREAM - see Section 4.5 below) and output (subroutine POPOUT).

(iii) If the GA termination conditions are met (Section 4.6), the FORTRAN GA terminates at this point, otherwise the parent
population array PAR is used to generate offspring array CHILD(ngene, nchild) as follows:

(iv) The generation counter (variable igen) is incremented.

(v) Tournament selection (function NEWPOP and subroutine TOURN) identifies chromosomes which will enter the reproduction process (i.e. the mating pool) and forms an entirely new generation in array CHILD(ngene, nchild) to contain these. Tournament selection is described in more detail in Section 3.2 below. The population size may be changed from npar to nchild at this stage (see Section 4.3 below for more details).

(vi) The offspring are created by successive cross-overs (subroutine XOVER), mutations (subroutine MUTATE) and inversions (subroutine INVERT) each applied separately and independently to the entire population in array CHILD, with altered genes/chromosomes simply overwriting the existing ones. There is no “elitism” or “preselection”, i.e. the fitnesses of the starting pool arising from tournament selection then play no further part in managing these reproduction operations and with random processes there is no guarantee that any parent will reappear in the offspring. The reproduction operations are described in more detail in Sections 3.3 and 3.4 below.

(vii) The resulting offspring in array CHILD(ngene, nchild) are then compared with each other and one of any duplicates (“clones” or “twins”) subjected to additional mutation so that all members should be unique (subroutine UNIQUE), though with random processes this cannot be guaranteed (see Section 4.1 below for further details). This could be avoided by commenting out the call to subroutine UNIQUE.

(viii) Finally, the offspring array CHILD are transposed into array PAR to become the parents of the next generation, and the process continues as above.
3.2 Tournament Selection

The form of s-member tournament selection implemented proceeds as follows (where s = number of members participating in the tournament):

A. There is a parent population array PAR(ngene, npar) with NPAR members having fitnesses in array PARFIT(npar) known from the simulation of the performance of each chromosome.

B. Each s-member tournament round proceeds until the required mating pool of NCHILD offspring in array CHILD(ngene, nchild) has been generated (at which point it terminates, whether all parents have competed or not).

(i) Usually NCOMP (a variable declared by user in range such that 2<=ncomp<npar) members are chosen randomly from those members of the parent population from array PAR who have not yet been chosen in this current round (other than the last group remaining at the end). Towards the end of the round there may not be enough parents not previously selected to give NCOMP competitors, so the last one or two groups may be adjusted in number to give roughly equal groups as close as possible to the desired NCOMP while ensuring that the entire parent population has been involved at least (but only) once (unless a very small offspring population with nchild<npar is completed before they have had a chance).

(ii) With each s-member group, their fitnesses (from array PARFIT) are compared and usually the NWIN (a variable declared by the user in range 1<=nwin<ncomp) most fit are added to the mating pool in array CHILD. If NCOMP is varied in the last two groups as above, then the number of winners is adjusted proportionately to ensure consistency in likelihood of selection.

C. If an s-member tournament round is completed because all parents have competed, but the offspring population is not yet completed, then further such rounds are initiated until there are the required NCHILD members of the offspring pool.
3.3 Cross-over

The form of cross-over implemented proceeds as follows:

A. Selection has produced a mating pool of NCHILD offspring in array CHILD(ngene, nchild), some of which will be modified (overwritten) by the reproduction operations (including cross-over).

B. Cross-over proceeds through the pool until all (or all but one, as one member cannot be crossed on its own) members have participated (with or without actual cross-over depending on the probability of cross-over).

(i) NCROSS (a variable declared by the user in range 2<=ncross<nchild) members are chosen randomly from those members of the population in array CHILD who have not yet been chosen in this current round (other than the last set <=ncross which defines itself - unless it is only one member).

(ii) If a random number is greater than the probability of cross-over PCROSS (a variable declared by the user in the range 0<=pcross<=1.0) then cross-over is not performed and this group is left unchanged, otherwise it is altered as follows.

(iii) NPOINT (a variable declared by the user in range 1<=npoint<ngene - the number of genes in each chromosome) gene locations are picked at random:

- cross-over (exchange of genes between members of group) does not occur from gene 1 to the first location, and

- then cross-over does occur from just after the first location to the next or the end of the chromosome, and so on, depending on the number of locations specified (NPOINT).

(iv) Genes between these limits are cycled around the chosen chromosomes in the order that they were randomly picked:

- those from the first go to the second, etc,

- until those from the last go to the first.

For example, with ngene = 8, ncross = 2 and npoint = 2, then starting with:
Assuming for illustration that the random number selected is <=pcross and for npoint = 2, locations of cross-over points are randomly picked to be 3 (thus cross-over site C1 = 4) and 7 (thus crossover site C2 = 7), then the two-point cross-over operation gives:

<table>
<thead>
<tr>
<th>gene location</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>first randomly chosen member</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>second (ncross) randomly chosen member</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
</tr>
</tbody>
</table>

Genes in positions 4 to 7 (inclusive) have been interchanged between the two members.

As a second example, with ngene = 6, ncross = 4 and npoint = 1, then starting with four members picked at random and in random order (say members 79, 107, 151 and 23 out of 200):

<table>
<thead>
<tr>
<th>gene location</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>chromosome 79</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>chromosome 107</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>chromosome 151</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
</tr>
<tr>
<td>chromosome 23</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
</tbody>
</table>

Assuming for illustration that the random number selected is <=pcross and the one-point npoint = 1 location of the cross-over point is randomly chosen as 3 (C1 = 4 below), then the one-point cross-over operator gives:
<table>
<thead>
<tr>
<th>gene location</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>chromosome 79</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>chromosome 107</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>chromosome 151</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>chromosome 23</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>P</td>
<td>Q</td>
<td>R</td>
</tr>
</tbody>
</table>

3.4 Mutation and Inversion

Unlike cross-over (Section 3.3 above), the other two operations of mutation and inversion are single-chromosome operations. Each is applied independently and the entire population is considered as follows:

A. Selection and then cross-over (and in the case of inversion, then mutation) has produced a pool of NCHILD members in array CHILD(ngene, nchild), some of which will be modified by the single-chromosome operators (mutation then inversion separately and independently).

B. In both cases, the entire population is scanned one chromosome at a time. For each chromosome, if a random number is greater than the user-declared operation probability (PMUT for mutation, PINVRT for inversion) then the chromosome is left unchanged, otherwise it is altered (by overwriting) by means of the operator (mutation or inversion).

Each of the three reproduction operators (cross-over, mutation, inversion) is applied independently. With random selection it is possible that individual chromosomes may be subjected to none, or all three, or any combination of one or two of these.

Mutation in this implementation is a gene-operator, whereas inversion is a chromosome-operation. For each chromosome selected randomly for mutation (based on the probability of mutation PMUT), a single gene in the sequence (ie range 1 to NGENE) is chosen at random and modified by user-defined code in subroutine MUTGEN to allow case-specific forms of gene mutation. Note that, as implemented, the probability of mutation applies to the chromosome and not the genes and that therefore only one mutation per chromosome may occur. It
would be possible to include an additional gene loop in subroutine MUTATE to allow the probability to be applied to individual genes, which would create the possibility that more than one mutation could occur in a chromosome (in this case subroutine MUTATE would call MUTGEN directly, omitting MUTCHR).

Inversion proceeds as follows:

(i) For each chromosome selected randomly for inversion (based on the probability of inversion PINVRT), two separate gene locations are randomly selected (i.e. in range 1 to NGENE). These may be in any order (i.e. low-high or high-low) as the sites are randomly chosen.

(ii) The order of genes between and including these two locations is then reversed using the first one chosen as the starting point and the second as the end point.

For example, with ngen = 12 the chromosome randomly selected for inversion is:

<table>
<thead>
<tr>
<th>gene location</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>chromosome genes</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
</tbody>
</table>

Assuming for illustration that the inversion locations are randomly chosen as 4 (S = start of inversion) and 10 (E = end of inversion) (in that order) then the result of inversion would be:

<table>
<thead>
<tr>
<th>gene location</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>chromosome genes</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>J</td>
<td>I</td>
<td>H</td>
<td>G</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>K</td>
<td>L</td>
</tr>
</tbody>
</table>

Thus the bits between S and E have their order reversed.

Assuming instead that the same two inversion locations were chosen, but due to randomness in the reverse order, i.e. 10 (S) and 4 (E), then the result of inversion would be instead:
Thus the genes enclosed by $E$ and $S$ are not inverted.

Note that where an odd number of genes is to be inverted (e.g. 7 in the above examples), then the “centre” gene will remain unchanged by the reversal (i.e. 7 = G and 1 = A in the above, respectively).

## 4 GA OPERATION OPTIONS

### 4.1 Selection and Reproduction Operations

The heart of any GA is the set of operators used in generational progression:

(i) selection (here by an s-member tournament),

(ii) reproduction (here by cross-over, mutation and inversion),

(iii) preservation of population diversity (here by removing duplicates, and also, discussed separately below, refreshing the population by incomers i.e. immigration).

Control of these in the FORTRAN GA software is essentially by three means:

(a) setting control parameters in DATA declarations in subroutine GENALG;

(b) commenting-out (by putting a C in first column of any line) or replacing particular statements (principally in subroutine GENALG);

(c) writing case-specific code for a particular application, especially for the generation of new chromosomes and the mutation of genes(subroutines NEWCHR and MUTGEN).

For these principal processes (described in more detail in Section 3 previously):

A. Selection by s-member tournament:
The normal tournament is “binary” with two randomly-selected competitors and one victor in each round (each competitor participating only once). By altering the DATA declaration for NCOMP (2 <= ncomp < population size) the number of competitors in each round can be varied, as can the number of victors with the variable NWIN (1 <= nwin < ncomp).

B. Reproduction by cross-over, mutation and inversion:

These can be set to occur with probabilities from 0.0 (never) to 1.0 (certain), with typical values from the literature (Davis 1991, Davidoar 1991) of:

- cross-over: pcross = 0.6 to 0.8 (depending on the number of members in the tournament). The value may need to be quite low say 0.1 to 0.3 for s > 2).
- mutation: pmut = 0.001 to 0.01
- inversion: pinvrt = 0.0 to 0.01

If pmut or pinvrt are set to 0.0 (e.g. not all implementations use inversion, as in Goldberg 1989) then it may be better to comment-out the calls to subroutines MUTATE or INVERT in subroutine GENALG. With inversion there are no further options. With cross-over there is a choice of number of cross-over points NPOINT (1 <= npoint <= length of chromosome) and of the number of chromosomes involved in each cross-over NCROSS (2 <= ncross < npop). Usually npoint = 1 or 2 (with ncross = 2 for pairs of chromosomes) or npoint = (L-1) for uniform cross-over (where L is the length of the chromosome). These variables and the probabilities are also set in DATA declarations in subroutine GENALG. With mutation, the literature contains many possible variants and it may necessarily be case-specific. It is necessary to write case-specific code in subroutine MUTGEN to accomplish this.

C. Preservation of population diversity by altering duplicates:

Davis (1991) recommends this procedure in certain situations; Goldberg (1989) does not mention it and it has not normally been used in past studies at the University of Adelaide. It can be avoided by commenting out the call to subroutine UNIQUE in subroutine GENALG. If it is retained then two points to note are:

(i) If it uses randomly-based alterations to chromosomes (e.g. mutation as implemented) there can be no absolute guarantee that this will not
create a new member duplicating another existing one (see, e.g., the example in Section 5 and Appendix C).

(ii) As implemented (Appendix A), mutation (subroutine MUTCHR) is used to change any duplicate chromosome found. Subroutine UNIQUE contains commented-out alternative lines that would enable inversion (subroutine INVCHR) or the random generation of a completely new chromosome (subroutine NEWCHR) instead (subroutines MUTCHR, INVCHR and NEWCHR have the same parameter structure). Mutation and inversion have the advantage of preserving some of the genetic information of the chromosome to be altered, but the disadvantage is that they introduce more of whichever operation than defined by their probability of occurrence. Completely new chromosomes do not have this effect and increase population diversity with new information.

4.2 Chromosome Size

The chromosome size (i.e. number of genes) is fixed at the value NGENE declared in subroutine GENALG (warning: this may affect local declarations in subroutines XOVER and possibly SIMULA, depending on details of case - see Section 2.4 above).

4.3 Population Size

There are two basic population management choices to be made:

(i) Fixed or varying population size? - discussed in this section below.

(ii) No interference with GA-generated population (other than that described above to avoid duplicates) or replacement of some population members (with which there is a further choice of continuously replacing low-fitness members or periodically “refreshing” the population with new members) - to be discussed in Section 4.4 below.

For (i) above:

A. Fixed population:

The population is fixed at the value NPAR passed as a parameter to subroutine GENALG. Values of MAXPOP and MINPOP declared in subroutine GENALG are not important other than that maxpop must be \( \geq \) npar. Variable population is disabled by declaring FAVG = 0.0 and FPK = 0.0 (with sensible values for dpk = 0.0, eavg = 1.0 and epk = 1.0) in subroutine GENALG.
B. Variable population:

The initial population size starts at the value NPAR (the number of selected parents) passed as a parameter to subroutine GENALG and thereafter may vary between the limits MINPOP and MAXPOP (\(\geq\) npar) declared in subroutine GENALG. It makes sense to start with the number of parents (npar) relatively low. Variable population is enabled by setting FAVG > 0.0 and/or FPK > 0.0 (with appropriately chosen values for EAVG, DPK and EPK, e.g. as illustrated in Appendix A). The tournament selection (subroutine TOURN) actually implements the population change, with any new members being descendants of the parent population (Section 3.2).

In the implementation shown, two effects based on the population fitness statistics (calculated in subroutine POSTATS) are deemed to influence population size by biological analogy (in function NEWPOP):

(i) *As the average level of population fitness increases, a population is likely to grow:*

This is implemented by comparing both the average and median at the current and immediate past generations:

\[
x = \frac{\text{(current average)}}{\text{(past average)}} - 1 + \frac{\text{(current median)}}{\text{(past median)}} - 1
\]  

\(4.1\)

The average considers the effects of all members of the population; the median considers how these are distributed. The population size is incremented by the factor (after the second generation only):

\[
1 + \text{FAVG} \times x^{\text{EAVG}}
\]  

\(4.2\)

(coded appropriately to avoid failure conditions) where FAVG and EAVG are declared by the user in subroutine GENALG.

(ii) *As the population diversity decreases, to avoid inbreeding (or premature convergence), more members are needed to restore diversity:*

This is implemented by looking at the distribution for the current distribution:

\[
x = -1 + \frac{\text{(current median + DPK \times current mean absolute deviation)}}{\text{(current peak fitness)}}
\]  

\(4.3\)
The median and mean absolute deviation are preferred here to the average and standard deviation (both available) as more robust statistics for non-normally distributed populations (Press et al 1992). The population is incremented by the factor:

\[ 1 + \text{FPK} \times x^{\text{EPK}} \]  

(4.4)

(coded appropriately to avoid failure conditions) where DPK, FPK and EPK are declared by the user in subroutine GENALG.

Warning: any changes to population size variables NPAR, MAXPOP and MINPOP affect local declarations in subroutines TOURN and XOVER (see Section 2.1 above).

### 4.4 Deliberate Changes to Population Members

The implementation may deliberately change population members to avoid duplicates (using subroutine UNIQUE as described in Section 4.1 above) and/or in addition as a result of a deliberate user strategy (through subroutines REFRESH and RMSIMR to be described in this Section). Three alternatives for the latter are available and the last two can be implemented simultaneously:

A. **No interference with population:**

   This is the **standard** approach and is achieved by disabling the alternatives below by setting NFRESH > MAXGEN (e.g. = 1,000,000) and PFRESH = 0.0 and PFEASI = 0.0 (with both FFRESH and FEASIB = some low value) in subroutine GENALG declarations.

B. **Periodic population refresh:**

   To avoid premature convergence with small populations, one approach is to periodically (every NFRESH < maxgen generations) cull less fit members (with a low or “lethal” fitness < FFRESH) at a probability PFRESH > 0.0 (e.g. = 1.0) and replace them with randomly generated new chromosomes. This is enabled by declaring appropriate values for NFRESH < maxgen, PFRESH > 0.0 and FFRESH in subroutine GENALG declarations. Subroutine RNSIMR will keep generating new members and evaluating their fitnesses until they satisfy the required conditions (the total number of simulations will be recorded as well as the number of chromosomes chosen to be replaced). Setting FFRESH too high will lead to a large number of simulations until a satisfactory population is achieved.
C. Replacement of infeasible chromosomes:

To eliminate infeasible (e.g. physically impossible) chromosomes the user could either code a fitness penalty function in the case-specific subroutine SIMULA or use this replacement option. This will, at every generation, cull members with an infeasible fitness < FEASIB at a probability PFEASI (= 1.0 normally for this application) and replace them with randomly generated new chromosomes, with subroutine RNSIMR evaluating their fitnesses until they satisfy the required conditions (as above). It is enabled with any appropriate values for PFEASI > 0.0 and FEASIB in subroutine GENALG declarations.

For the last options B and C above, completely new chromosomes are generated by subroutine NEWCHR. As alternatives, commented-out lines in subroutine RNSIMR suggest that mutation or inversion could be used to adapt the unfit members while preserving some of their genetic inheritance (but increasing the incidence of the operation above the stated probability).

4.5 Record of Best Chromosomes

The FORTRAN GA software uses subroutine CREAM to record the NCREAM \(\geq 1\) (declared in subroutine GENALG) best chromosomes from all generations. This can be avoided by commenting-out the call to CREAM in subroutine GENALG (and if so the initialisations in subroutine INIPOP can also be commented-out). See also Section 4.6 (iii) below.

4.6 Number of Generations and Convergence Criteria

As implemented, three convergence criteria are used in the FORTRAN GA, any one of which will stop the process:

(i) number of generations evaluated equals the maximum number of generations allowed = MAXGEN > 1 (declared in subroutine GENALG); or

(ii) current population statistics show convergence to a narrow band of fitnesses with:

(standard deviation of fitnesses of the current generation) \(< = \text{SMALL}^*\) (maximum - average)

(could be re-coded to suit user) where SMALL is declared as DATA in subroutine GENALG; or
(iii) no additions to CREAM (record of best chromosomes found) for three successive generations (comment this out if subroutine CREAM commented out as in Section 4.5 above).

4.7 Random Number Seed

If it is desired to change the initial population for any given problem, then it is necessary to redefine the values of SEED and/or START in function RANDOM. For GA experimentation it is often desirable that the same seed population be used for successive optimisations and it is an advantage of the Knuth random number algorithm used (Press et al 1992) that it gives the same sequence of random numbers for the same seed (Goldberg 1989).

5 AN EXAMPLE APPLICATION

5.1 What is Necessary to Use this Code for a Specific Application?

For an illustration, the problem of maximising $x^2$ where $x$ is an unsigned 5-binary-bit positive integer, as used by Goldberg (1989), is demonstrated. This is a trivial application - most real applications will generate more code than the GA itself. However this example illustrates what steps are necessary in order to implement the GA for any other specific application.

In this section, discussion specific to Goldberg's $x^2$ example is italicized.

The user must adapt or supply code as follows:

(i) The software is provided as a subroutine GenAlg (Appendix B.1) and a user-written main program is required to call GenAlg (Section 2.5).

(ii) GAs are generic and problem-independent. The user must provide:

- a problem simulation code,
- to which data is provided by de-coding a chromosome and
- whose results are interpreted and encoded into a fitness, through an interface to subroutine GenAlg defined at subroutine SIMULA (Appendix B.2).

(iii) The form of the chromosomes is determined by subroutine GenAlg, i.e. vectors of real numbers in a pre-determined order. However, the processes of:

- creating a new chromosome (subroutine NEWCHAR), and
- mutating a gene within a chromosome (subroutine MUTGEN), will depend on how the user implements (ii) (Appendix B.2).
(iv) Within the general purpose GA subroutine GenAlg, the user needs to set data declarations in order to (Section 4):

- define the GA operation options to be implemented,
- set out the memory requirements for the particular application, and
- establish what data recording and output is desired.

### 5.2 The User-Defined Calling Main Program

*The minimum program necessary is illustrated for the Goldberg \( x^2 \) example at the start of Appendix C.*

At the very least this main program must call subroutine GenAlg (see Appendix B.1) with data supplied for the single integer npar in subroutine GenAlg:

\[
\text{npar} = \text{positive integer defining the starting size of the chromosome population (or simply population size if this is not to be varied) - see Sections 3.1A and 4.3.}
\]

*\( i.e. \) inpop = 4 for Goldberg (1989) \( x^2 \) example in Appendix C*

In practice the main program may provide data input/output for the problem under investigation. Such data will need to be made available to the simulation code through COMMON declarations.

### 5.3 The Simulation Code Interface at Subroutine SIMULA

*The example code for subroutine SIMULA in Appendix A is for the Goldberg (1989) \( x^2 \) example:*

- *It decodes each chromosome to be run, i.e. pop (i) from a positive binary integer into a positive real number x (high order bit first in chromosome), and*

- *no further simulation being necessary for this trivial example,

- *it encodes the corresponding fitness popfit (i) as \( x^2 \).*

Data is passed to subroutine SIMULA through the parameters defined in Appendix B.2. Subroutine SIMULA must contain an outer loop to run the simulation only for the nrun specified cases indexed in pointer vector lrun, eg (from Appendix A example):

\[
\text{DO 1302 k = 1, nrun}
\]

\[
i = \text{lrun(k)}
\]
etc

1302 CONTINUE

All other data required by the decoding, simulation and encoding for any particular case would have to be declared locally or passed from the main program (Section 5.2 above) by COMMON declarations. It is likely that SIMULA will call other software provided for the simulation and it will have to ensure that all the necessary parameters for this are defined. Parameter array sizes in subroutine SIMULA are dynamic and controlled by the declarations in subroutine GenAlg (see Section 5.5 below). Other array size declarations for additional code must conform to these.

5.4 Case-Specific Subroutines NEWCHR and MUTGEN

For the Goldberg (1989) $x^2$ example, subroutines NEWCHR and MUTGEN set up “genes” as either 0.0 or 1.0 to mimic binary bits, while SIMULA uses these to evaluate $x$ and hence the fitness $x^2$. In this FORTRAN implementation, genes are real numbers in the range 0.0 to 1.0 and there are no “allele” or bits from which genes are constructed (Section 2.2i). However it is quite easy to restrict genes to the values 1.0 (simulating a binary bit 1) and 0.0 (simulating a binary bit 0), with the random number generator in function RANDOM used for new “bits” as follows:

\[
\begin{align*}
\text{IF} \ (\text{Random()}.LT.0.5) & \quad \text{THEN} \ \text{gene} = 0.0 \\
& \quad \text{ELSE} \ \text{gene} = 1.0 \\
& \quad \text{ENDIF}
\end{align*}
\]

Any other discrete number representation (e.g. octal, hexadecimal) can be constructed in a similar way, as the Knuth random number algorithm adopted gives uniform random deviates (Press et al 1992). It is then necessary for subroutine SIMULA to decode these “bits” into a real number (high order bit first in chromosome). Obviously it is rather wasteful of memory to use real numbers to encode “bits”, but the information content of this example is so trivial that a GA can only operate on the richest possible chromosome structure.

Subroutine NEWCHR(ngene, CHROM) creates a new chromosome CHROM (ngene) of NGENE genes. It must contain an outer loop for the ngene genes as in the example of Appendix A:

\[
\begin{align*}
\text{DO} & \quad 1101 \ i = 1, \ \text{ngene} \\
\text{1101} & \quad \text{CONTINUE}
\end{align*}
\]

In the $x^2$ example genes are simply set as “bits” by the device above.
Real number genes (in the range 0.0 to 1.0) can be given simply by function RANDOM, e.g.:

\[
\text{CHROM}(i) = \text{RANDOM}()
\]

Subroutine MUTGEN(GENE) takes a real number GENE (in the range 0.0 to 1.0) and transforms it. This transformation may depend on:

(a) The particular problem and hence form of chromosomes and genes under consideration:

\textit{e.g., the } x^2 \textit{ example in Appendix A simply shows a change of "bit" from 0.0 to 1.0 or vice versa.}

(b) The type of mutation to be implemented:

A variety of different mutation operators have been defined in the literature, \textit{e.g.} “bitwise” \textit{(as in the } x^2 \textit{ example)}, “creeping” or “adjacency”, etc. Subroutine MUTGEN leaves the user free to implement whichever form seems most appropriate to the problem under consideration.

5.5 Summary of Option and Case Declarations in Subroutine GENALG

5.5.1 Selection of GA options (Section 4)

(a) GA s-member tournament selection (Section 3.2):

Set DATA declaration in subroutine GENALG for:

\[
\begin{align*}
\text{ncomp} & = \text{integer number of competitors in each tournament round} \ (2 < = \text{ncomp} < \text{npop}) \\
& = 2 \text{ in Appendix A for the } x^2 \text{ example ("binary tournament")}
\end{align*}
\]

\[
\begin{align*}
\text{nwin} & = \text{integer number of vectors in each tournament round} \\
& (1 < = \text{nwin} < \text{ncomp}) \\
& = 1 \text{ in Appendix A for the } x^2 \text{ example}
\end{align*}
\]

(b) GA reproduction by cross-over, mutation and inversion (Section 4.1):

Set DATA declaration in subroutine GENALG for:

\[
\begin{align*}
\text{ncross} & = \text{integer number of chromosomes involved in each cross-over, usually 2 for pairs} \\
& = 2 \text{ in Appendix A for the } x^2 \text{ example with a small population}
\end{align*}
\]
npoint = integer number of cross-over points along chromosome, usually 1 or 2 or (ngene-1) where ngene = length of chromosome
= 1 in Appendix A for the $x^2$ example with a short chromosome

pcross = real probability (never 0.0 <= pcross <= 1.0 certain) that cross-over occurs with selected across chromosomes
= 1.0 in Appendix A as in $x^2$ example

pmut = real probability (never 0.0 <= pmut <= 1.0 certain) that mutation occurs (case specific as in Section 5.4 above)
= 0.001 in Appendix A for the $x^2$ example illustration

pinvrt = real probability (never 0.0 <= pinvrt <= 1.0 certain) that inversion occurs
= 0.01 in Appendix A for illustration

(c) GA population control (Sections 4.3 and 4.4):
Set DATA declaration in subroutine GENALG for:

(i) population size variation? (Section 4.3):
    - no: declare real favg = fpk = 0.0 (with dpk = 0.0, eavg = epk = 1.0)
    - yes: declare real favg /= 0.0 and fpk /= 0.0 with corresponding appropriate values for real dpk, eavg and epk.

    e.g. example in Appendix A shows that variable population size is activated with favg = 0.2, eavg = 0.15, fpk = 0.4, dpk = 1.33, epk = 0.7

(ii) deliberate replacement of population members? (Section 4.4):
    - no: set integer nfresh > maxgen (see Section 5.5.2 below) with real pfresh = pfeas = 0.0 (with ffresh and ffeas = 0.0).
    - yes: (a) periodic population refresh:
      Every integer nfresh > 0 generations, cull less fit members with fitnesses < ffresh at a probability pfresh (never 0.0 <= pfresh <= 1.0 always).

      (b) replace infeasible chromosomes at any generation:
      In every generation cull the members with an infeasible or “lethal” fitness < feasib at a probability pfeas (= 1.0 normally) (never 0.0 <= pfeas <= 1.0 always).
e.g. example in Appendix A for $x^2$ example disables this by setting
nfresh = 1000000 (with ppfesh = ffresh = pfeasi = feasib = 0.0).

5.5.2 Memory Requirements

(a) GA convergence control (Section 4.6):

Set PARAMETER and DATA declarations in subroutine GENALG for:

maxgen = integer maximum number of generations required

= 5 in Appendix A for $x^2$ example of Appendix C

small = real range multiplier for convergence criterion before
maxgen generations

= 0.15 in Appendix A for $x^2$ example

(b) Chromosome size (Section 4.2):

Set PARAMETER declaration in subroutine GENALG for:

ngene = integer number of genes in each chromosome

= 5 in Appendix A for 5-bit integers of Goldberg (1989) $x^2$
  example

(c) Population Size (Section 4.3):

Set PARAMETER and DATA declarations in subroutine GenAlg for:

npar = integer parameter to subroutine GenAlg set in main
program - see Section 5.2 above:

= 4 in Appendix C example

maxpop, minpop = integers defining maximum (maxpop >=
npar) and minimum (minpop< = npar)
  population size

e.g. in Appendix A for $x^2$ example maxpop = 8 and minpop = 3 for
Goldberg (1989) $x^2$ example of Appendix C.

(d) Number of “Best” Chromosomes Stored for Output:

Set PARAMETER declaration in subroutine GENALG for:

ncream = integer number (> = 1) of exclusive best (fittest)
  chromosomes found throughout all generations

= 4 in Appendix A for example of Appendix C
See also warning in Sections 2.1 and 4.2

5.6 Description of Appendix C Output for the $x^2$ Example

The declarations (Section 5.5 above) shown in Appendix A are for the very simple Goldberg (1989) $x^2$ example and are not representative of normal GA practice:

(i) In this example there are only a small number $ngene = 5$ of genes in the chromosome (actually allele “bits” of either 0.0 or 1.0), the population is restricted to the range $minpop = 3$ to $maxpop = 8$ (starting at the parameter $npar$ given as input = 4 in the calling main program of Appendix C) and only the $ncream = 4$ best chromosomes will be recorded over a maximum of $maxgen = 5$ generations. These values are much small than would normally be the case.

(ii) With such a small population, tournament selection can only be set up in the “standard” binary form with $ncomp = 2$ competitors in each round and the fittest competitor or victor selected to go into the mating pool with $nwin = 1$. Typical probabilities of mutation ($pmut = 0.001$) and inversion ($pinvrt = 0.01$) are used, with a probability of crossover ($pcross = 1.0$) assigned as in Goldberg (1989) for this example. With such a short chromosome ($ngene = 5$) only one-point crossover ($npoint = 1$) is used (with a longer chromosome $npoint = 2$ might be favoured) and, with such a small population, crossover is in standard pairs ($ncross = 2$). Note also that “bitwise” mutation is chosen for this application, as set up by subroutine MUTGEN (Section 5.4 above and Appendix A).

(iii) It is not necessary to avoid infeasible chromosomes in this kind of application so this option is eliminated with $pfeasi = 0.0$ (and $ffeasi = 0.0$). With such a small number of combinations ($2^5 = 32$) in this illustration it is not realistic to periodically “refresh” the population, so this option is eliminated with $nfresh = 1000000$, $pfresh = 0.0$ (and $ffresh = 0.0$). However, though it is not really appropriate for such a case, for illustration the variable population size option has been activated by setting both $favg = 0.2$ and $fpk = 0.4$ (not equal to zero) with $eavg = 0.15$, $dpk = 1.33$ and $epk = 0.7$ (these are not necessarily optimum values).

Appendix C contains the necessary calling main program (Section 5.2 above) and the output as generated by the code in Appendix A. In this example no warnings are generated for locally declared array bounds which are exceeded, so (other than the last line - from main program) output is produced only from four subroutines:
(i) GENALG:
The GA option set up is summarised at the head of the output. For repeated trials the lines in subroutine GENALG producing this summary could be commented-out.

(ii) INIPOP:
The randomly generated (and hence naturally different to that in Goldberg 1989) initial "seed" population is presented. Again, in practical cases this information may not be required and the lines in subroutine INIPOP producing it could be commented-out.

(iii) POPOUT:
Each successive generation is presented in full with its fitness statistics. Normally this would produce excessive output and the call to subroutine POPOUT in subroutine GENALG would be commented-out.

(iv) RESULT:
GA and population statistics, along with the best chromosome, are summarised for each generation (recorded in subroutine EVOLUT), then the best chromosomes found from all generations (recorded in subroutine CREAM) are summarised (in this case the four best chromosomes are recorded nream = 4).

The output shows the progress of the GA. The initial random population has a very similar average fitness to Goldberg's example (1989). The first generation is of the specified initial size (4) and being randomly initialised, has not been formed by any cross-over, mutations or inversions. Without replacements (throughout, as these options inactive), the number of simulations at each generation simply equals the population size. Obviously, of the first generation, all its different members are better than any previous (none initially), except for the "lethal" with fitness = 0.0. As the generations evolve, fewer are better than those found previously. For the second generation, reproduction has been by two cross-overs (so all its members have been crossed) and one mutation. No inversions at all occur because of the low probability of occurrence. The mutation probability is even lower, but a number do occur when the code tries to eliminate duplicates in the population (subroutine UNIQUE). However, in this example with such a small number of combinations ($2^5 = 32$), it is almost inevitable that mutation will not eliminate all duplicates. These occur in the 5th generation (fitness = 784.00). The population size occasionally increases (generations 3 and 5, due to improving average fitness and for generation 5 a maximum close to the median) and decreases (generation 4, due to decreasing average, coupled with a maximum fitness well above the median). The four best chromosomes are found, almost inevitably in this example case, with the second best (fitness = 900.00) appearing by generation 2 but the best (fitness =
961.0) not emerging until generation 5 in this trial after a total of 20 evaluations of the fitness function.

6. **AVAILABILITY OF AND RESPONSIBILITY FOR USING THE CODE**

By presenting the code in this format it is clearly intended that it should be freely available to the educational, scientific and engineering community. However, this does not mean that the author or the University of Adelaide and the University of Newcastle upon Tyne, U.K. relinquish their rights to it. ALL RIGHTS ARE RESERVED. Use of all or part of this code without licence is welcomed on the understanding that it implies acceptance by the user of the conditions that this:

(i) MUST only be for the purposes of non-profit making and non-commercial education or research where the results will be made freely available in the public domain and that such use will be acknowledged in all reporting of that work; and

(ii) must NOT be used for any commercial purposes or for research or consulting which will not be published or made generally available or for financial gain to any individual(s) or organisation(s).

Distribution of all or part of this report and/or code is welcomed on the understanding that both distribution and receipt imply full acceptance of the conditions above and of the continuation of these conditions.

Use of this report and/or all or part of the code therein implies acceptance that is with NO WARRANTY and NO LIABILITY on the part of the authors or of the Universities of Adelaide or Newcastle upon Tyne. No claims are made, in respect of either the report or the code, for their correctness, accuracy and reliability or compatibility with any other software or computing environment. Use of all or part of this code implies the acceptance that it is at the user’s own risk and all liability for any resulting direct or consequential damages rests with the user.

Finally, the code (with this accompanying description) is intended to offer a simple yet adaptable and portable introduction to GAs. No claims are made for its computational efficiency or the elegance of its constructions or its exhaustiveness of the possibilities of GA techniques.
7. CONCLUSIONS

A general purpose Genetic Algorithm (GA) code has been written in Fortran to facilitate wider use of GAs in engineering applications as well as further research on GAs. Its general features are:

(a) a “simple” GA using generational reproduction without “elitism”; based on
(b) gene representation restricted to real numbers in the range 0.0 to 1.0 (through more complex structures can be simulated at the cost of inefficient use of memory); and
(c) chromosomes restricted to fixed length (number of) and location of genes; with
(d) reproduction operators (cross-over, mutation, inversion) which act on the population independently of each other (and inversion implemented in addition to cross-over and mutation).

It contains a number of innovative features and options, designed to enable further investigations of GA strategies:

(i) s-member Tournament selection with any numbers of competitors and victors in each contest (including the conventional “binary” tournament);
(ii) “rotating” cross-over, with any (currently fixed) number of chromosomes in each cross-over pool (including the conventional pairs);
(iii) elimination of population duplicates before simulation;
(iv) adaptive variation of population size;
(v) optional population management, either eliminating “lethal” or infeasible chromosomes or/and “refreshing” the population with “incomers” every few generational cycles; and
(vi) the ability to record a user-set number of most-fit chromosomes to assist the user in analysing the solution space.

The primary objective of the software is to support further joint research between the Universities of Adelaide and of Newcastle upon Tyne. The primary objective of this report is to make the GA software generally available to the engineering, educational and scientific communities for public-domain research and development and, to this end, the FORTRAN coding has tried to avoid using system - or machine-dependent operations.
8. REFERENCES


APPENDIX A. CODE LISTING

This contains the code listing set up (declarations and case-specific routines) for the illustrative example of Appendix C. Module descriptions are given in Appendix B.

The Fortran code is available via anonymous ftp from

ftp.aelmg.adelaide.edu.au

/pub/civeng/codes/fortranga.f
null
Avoid by setting pmut=0.0 or commenting MUTATE call out.

NBR: mutation is not exclusive of cross-over and/or inversion
and additional mutations may also occur as a result of
randomly reducing the number of elements in the population duplicates - see NBR Risk of Mutation
Standard is pmut = 0.000 to 0.01.

Set up below with pmut=0.001 (and in subroutine HUGEN).

7. Inversion at chromosome probability PINVNT. Avoid by setting

innvnt=0.0 or commenting INVERT call out. NBR: inversion is not

exclusive of cross-over and/or mutation.

Standard is innvnt = 0.0 to 0.01.

Set up below with innvnt = 0.01.

8. After reproduction (cross-over, mutation and inversion) then

population checked for duplicate chromosomes and these

are refined by additional mutations (see 5. above) or inversion

or entirely new 'incomer' chromosome as in 6. and 5. above.

- see commented out alternative lines in subroutine UNIQUE to

avoid redundant simulations and to ensure diversity, NBR: as

mutation etc are random there is no absolute guarantee that no

duplication will remain after this process.

Avoid by commenting out call below to subroutine UNIQUE.

Standard is to avoid.

Set up below to occur.

INTEGER INODE, NIPM, NISCR, NPOINT, NIPREF
REAL FCROSS, PMUT, PINVNT, PFPREF, PAPER, PFES1, FEAS1
DATA norcross,nipm,niscr,apoint,nipref
1 2 1 2 1 1.000000/3
DATA apoint,nipref,papers,pref,precision,feas1
1 0.000000,0.01,0.0,0.0,0.0,0.0,0.0/3
REAL PFP, AVG, MEAN, PPRE, PFPK, EPNK
DATA PFP,avg,mean,fpk,epk,fpk
1 0.0,0.0,0.0,0.0,0.0,0.0,0.0/3
COMMON /FPBK/ fouw,eval,fpk,ek,fpk
C Set bandwidth SMALL for cross-over GA convergence condition
C and automatically Initialise convergence condition (CONVERGEC)
C and generation on the basis of generation or sequence of
C refresh and random number counter IRCALL (common block HNOM for
C refresh and random number counter IRCALL (common block HNOM for
C function RANDOM also keeps values in NODE (NS) table used to
C generate sequence of random numbers - see function Random.
C Set繫 SIMPLE.
C LOGICAL CONVERGE
C INTEGER ICNTR, IFREF, IRCALL
C DATA small,converge,ipm,prec,ir,icall
1 0.0,0.0,0.0,0.0/3
COMMON /RMDM/ icall, knuth
C Best screen chromosomes from all generations so far stored in
C CROMG (ngen,ngen,ngen) with corresponding fitnesses stored in FCROMG (ngen,ngen,ngen)
C and rank pointers in ascending order in LCRANK (ngen,ngen,ngen).
C A record of evolution of each succeeding generation is stored in
C EVOLG (ngen,ngen) (at each generation NPNBR population size, 82. number
C refreshed and/or replaced if feasible, 3- number of chromosomes =e, 82. number
C 4- number of mutations, 5- number of inversions, 82. number at the
C generation better than best of previous generations in cream.
C 7- total number of simulations done at current generation, with
C 8- generation statistics 1- minimum fitness, 8- fitness average (mean),
C 3- fitness median, 4- fitness standard deviation, 5- average (mean)
C absolute deviation from median, and also CROMG (ngen,ngen,ngen)
C CROMG (ngen,ngen,ngen) with index of rank pointers (in
C ascending order) CSFMR (ngen,ngen,ngen) produce HNOM offspring in
C HNOM (ngen,ngen,ngen).
C Each generation npar parents from PARENT (ngen,ngen,ngen) with
C fitnesses PARFNT (ngen,ngen,ngen) and index of rank pointers (in
C ascending order) CSFMR (ngen,ngen,ngen) produce HNOM offspring in
C HNOM (ngen,ngen,ngen).
or none better than cream for three generations - use of
max avoids accessing zero or negative xsubscripts:
  IF ((lgen .GE. maxgen) OR,
  (revol(4, lgen, lgen, small) )
  .OR.((revol(4, lgen, lgen, maxvol(lgen-1, lgen)) =
  revol(4, lgen, maxvol(lgen-2, lgen))))
  THEN
  SET termination condition and skip reproduction
  converge = .TRUE.
  ELSE
  reproduction occurs to produce offspring child so
  increment generation counter
  lgen = lgen + 1,
  get new population size depending on fitness statistics
  - pass columns of revol as statistics vectors and max avoids
    accessing zero subscripts first time
  NHCHILD = Newpop(1, npar, minpop, maxpop, revol(4, lgen-1, lgen),
  revol(4, lgen-2, lgen))))
  IF variable probabilities required - implement here (not done)
  Tournament forms basis of new population (with size change)
  CALL Tour(ncomp, nwin, ngen, npar, nhchild, parfit, parent, CHILD)
  New generation formed by cross-over, mutation and inversion
  CALL Xover(crecs, cross, cpoint, ngen, nhchild,
  IREVOL(4, lgen, pHILL))
  CALL Invert(crevnt, ngen, nhchild, IREVOL(4, lgen, CHILD))
  After reproduction, eliminate population duplicates with
  additional mutations (and update mutation counter)
  CALL Unique(ngen, nhchild, IREVOL(4, lgen, CHILD))
  Finally transpose child into parent for next generation
  npar = nhchild
  DO 13 J = 1, nhchild
  DO 12 K = 1, ngen
    parent(K, J) = child(K, J)
  CONTINUE
  12 CONTINUE
  13 CONTINUE
  ENDIF
  ANM2 FORTRAN WHILE-DI (goto 10 above and endif)
  GOTO 10
  ENDIF
  WHEN evolution ended output results
  CALL Result(lgen, maxgen, ngen, ncream, ccream, frcream, ccream, frcream,
  level, revol, cevol)
  RETURN
  ENDD
  End of subroutine GENHil

2. CONTENTS (index of GA subroutines and functions called by
   subroutine GENHil)
   *a. subroutine INITPOP - INITialize seed POPulation
   *b. subroutine REFRESH - REFRESH population with lincorns
   *c. subroutine RMSIRM - Run SIMulation with r-emplacement
   *d. subroutine CREAM - record CREAM of fittest chromosomes
   *e. subroutine POSTAT - get POPulation fitness STATistics
   *f. subroutine POPOUT - generate POPulation draws OUT-put
   *g. subroutine MEMPAT - calculate NEW POPulation size
   *h. subroutine SWAP - SWAP lincorn selection
   *i. subroutine XOVER - X(cross)-OVER a population
   *j. subroutine MUTATE - MUTATE a population
   *k. subroutine INVERT - INVERT a population
   *l. subroutine SUBR - SUBRoutines
   *m. subroutine UNIQUE - ensure all population members UNIQUE
   Subroutine UNIQUE - output final RESULTS after GA ends

SUBROUTINE INITPOP(npar, ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
                     ngen, ncream, frcream, PARENT, ngen, ncream, frcream, PARENT,
Appendix A

C and replaces cream values appropriately. Assumes prior checks that
C rank pointers in loops and kream are correctly set up on entry.
C modifies only KCREAM, FCREAM, KCREAM and LCREAM and not other
C parameters or global variables.
C Uses function CHANGE and subroutine ISRAKE

LOGICAL CHEAMS
REAL FITJOL DATA fittol (0.0)
C Fittol is tolerance for fitness comparisons where
C exact equality too demanding for equivalence
C (currently set at 0.0 for exact equality)
INTEGER KPPOP, I
C KPPOP counts no of current generation evaluated
C I is loop counter
LOGICAL SWAP, THERE
C SWAP is loop termination flag for replacement process
C THERE is termination flag for comparison
C Start
C Initialize counters for number of new additions to cream
C (kream and number of current generation done (kpopp)
C kream = 0
C Set termination flag for entire process (while-do loop)
SWAP = .FALSE.
C ANSI FORTRAN WHILE-DO (if labelled 1401 with goto below)
1401 IF (SWAP.EQ..TRUE.) THEN
C Cannot make more than the smaller of ncream or nppp swaps
C IF (kream.GE.ncream).OR.(kpop.GE.npop) THEN
C Set termination flag (maximum number possible)
SWAP = .FALSE.
ELSE
C Start with fitness in new generation (nppp-ncream next fitness, etc)
C and compare with least fit in cream (1 = then next, etc)
C IF (popfitlioprapop(kpop,kpop))
C .LT. Fcream(lcream+1.kcream))
C Set termination flag (none left better than cream)
SWAP = .FALSE.
ELSIF
C Candidate for inclusion in cream = check if already in
C cream for assumed non-existence in cream
C if there .FALSE.
C Initialize cream counter at least fit for repeat-until loop
C = 0
C ANSI FORTRAN REPEAT-UNTIL (labelled 1402 for IF-goto below)
1402 I = 1
C Check for fitness equality first - if fitnesses
C same than check for chromosomes equivalence
C IF (AAS(popfitlioprapop(kpop,kpop))
C .EQ. cream(lcream+1.kcream))
C there = kream(legepop,1,kpop) kpop(kpop,kpop)
C kream(lcream+1)
C Repeat until all cream done or terminated by there flag
C ANSI FORTRAN REPEAT-UNTIL (if-goto 1402 above)
C IF (I.LT.ncream).AND.(there.EQ..FALSE.)"GOTO 1402
C IF not is already in cream, then swap by overwriting
C if there.EQ..FALSE.)" THEN
C Swap fitness
C cream(lcream+1.kcream).POPFIT(popfitlioprapop(kpop,kpop))
C Swap chromosome (gene by gene)
C DO 1403 l = 1, kuream = 1, kcream = kuream + 1
C = pop(l,1,kpop(kpop,kpop))
C ANSI FORTRAN WHILE-DO (goto 1401 above and endif)
1403 CONTINUE
C Increment cream counter for this one added to cream
C kream = kream + 1
C ANSI FORTRAN WHILE-DO (goto 1401 above and endif)
C ANSI FORTRAN WHILE-DO (goto 1401 above and endif)
C ANSI FORTRAN WHILE-DO (goto 1401 above and endif)

C AA 21 vi 95
C (c)Copyright Alexander Anderson 1995. All rights reserved.
C No Warranty.
C
C At current generation number IGEN, output details of population
C POF(pop,mnpop) of NPOP chromosomes each with MGENE genes and
C fitnesses POPFIT(ind) with pointers in LPOPAF(ind) for use in
C assessing overall population fitness in statistics in PSTATS(5)
C (average, median, mean, standard deviation, maximum absolute
C deviation).
C Does not change any parameters or global variables.
C
C INTEGRER 1, 0
C Start
C
C PRINT 1, 'Population generation n, ngen'
C PRINT 1, 'Average Median Standard Mean Absolute'
C PRINT 1, 'Fitness Fitness Deviation Deviation'
C PRINT 1, '((pof(1,2,3,4)), pof(paste(1),1,2,3,4))'
C PRINT 1, '+pof(pop,fpop); j1,agen,agen'
C
C 1701 CONTINUE
C PRINT 1, '+pof(pop,fpop); j1,agen,agen'
C RETURN
C
C End of subroutine Popstat
C
C FUNCTION POFPOPLATION(pop, npop, MAXPOP, pasta, past)
C INTEGRER NPOP, NNEWPOP, MINPOP, HAXPOP, MAXPOP
C REAL PSTATS(5), PASTA(5)
C C Calculate new POF-population size for next generation
C AA 21 vi 95
C (c)Copyright Alexander Anderson 1995. All rights reserved.
C No Warranty.
C
C Given the last generation population size NPOP and the
C population size constraints MINPOP <= NPOP <= MAXPOP
C (defined with other information in common block POSIZE below)
C along with the generation statistics as follows:
C PSTATS(5) = current mean (average)
C PSTATS(2) = median
C PSTATS(5) = average (mean absolute) deviation (from median)
C
C corresponding past generation values in PASTA(5).
C
C Then calculates a new population size NNEWPOP. NPOP is:
C to trap zero values before division (possible in early cycles)
C and powers x^y written as x**(y-1) to avoid error
C raising possibly negative number to fractional power.
C Does not modify any parameters or global variables.
C
C REAL FAVG, EAVG, FPK, DPK, DPK
C COMMON /POSIZE/ POSIZE, MAXPOP, HAXPOP
C C FAVG, EAVG multiplier and exponent for change in average
C FPK, DPK multiplier and exponent with DPX number of
decimals for diversity change (maximum as outlined)
C REAL AVGFAV, DIVFAV
C adjustment factors for average and diversity also used
C for temporary storage
C REAL EPSILON
C DATA EPSILON /0.0000001/
C REAL EPSILON is max value to be raised to possibly negative power
C INTEGRER NPOP
C NPOP is temporary storage for population size
C Start
C
C First get effect of average and median - no change if stay the
C same. Population decreases/increases if they go down/up (i.e.
C less/more fitness gives smaller/larger population)
C

IF ((past(1,2,3,4,5) AND (past(1,2,3,4,5))) THEN
AVFAV = (past(1,2,3,4,5); past(1,2,3,4,5) * past(1,2,3,4,5));
IF AVGFAV < EPSILON THEN
AVFAV = 1.0 + AVGFAV
AVFAV = (AVFAV)**(EPSILON)
ELSE
AVFAV = 1.0
ENDIF
ELSE
AVFAV = 1.0
ENDIF
C Second get effect of diversity - if maximum line within/out
C (median-diviations) then want bigger/smaller populon to
C increase/decrease diversity (in ter to avoid overpopulation
C if (pstat(1,2,3,4,5)) THEN
DIFAC = (pstat(1,2,3,4,5); pasta(1,2,3,4,5));
IF DIFAC < EPSILON THEN
DIFAC = 1.0 + DIFAC
DIFAC = (DIFAC)**(EPSILON)
ELSE
DIFAC = 1.0
ENDIF
ELSE
DIFAC = 1.0
ENDIF
C Finally combine to get new size and check constraints
C HAXPOP = MAXPOP - EPSILON + EPSILON
IF (minpop <= HAXPOP) HAXPOP = minpop
IF (pstat(1,2,3,4,5)) RETURN
ENDIF
C End of function Newpop
C
C SUBROUTINE TOWN(pop, popw, popc, npar, schild, parfit, parent, CHILD)
C INTEGRER NPOP, MINPOP, MGENE, NPAR, CHILDD
C REAL PARTIT(npar), PARENT(npar, npar)
C CHILD(npar, npar)
C TOWNEnvironment for FA reproduction
C AA 21 vi 95
C (c)Copyright Alexander Anderson 1995. All rights reserved.
C No Warranty.
C
C Generates new population of CHILDD chromosomes CHILD(npar,schild) from
C existing population of NPAR chromosomes PARENT(npar, npar) where MGENE
C is chromosome length and PARFIT(npar) are the corresponding parent fit-
C cener's same index order as parent.
C
C using tournaments of NCOMP competitors giving MMN victors (assumes
C prior checks for npar x npar x npar win which ensures that no par-
C cent competes more than once in any one round during tournament.
C
C inlines CHILDD and other parameters or global variables
C
C uses randoms MHILDD and PARENT
C
C FRAMEWORK (maxpop = 1000)
C NMAXPOP (MAXPAR = 1000)
C NMAXPOP (MAXPAR = 1000)
C
C NMAXPOP (maxpar = 1000) is maximum population and is
C necessary to dimension list (below) - should be
C specified consistently in main program.
C
C INTEGRER KLEPT, NPOP, NRIGHT, NVIC, LIST(maxpop), 1, 0
C
C $KLEPT (<npar) is number of parent left in current round
C NRIGHT (<npar) is number of new child to current point
C NFIGHT (usually (npar) is number of competitors in current
C context)
C NVIC (usually -win -win is number of vicors in current context
C first k elements of LIST are indices of parent not yet
C competed, remainder have first npar of these are
C in current context.
C
C 1, J are general purpose loop counters
C 0, STOP, TEMP are temporary storage for sorting
C
C REAL TOT
C TOT is temporary storage for search
C Start
C
C Initialize competitive list with all spar parents at start (warm

Occasional typos and formatting issues are present in the text.
C if local bound maxpop exceeded - will probably corrupt memory
C if (nmax >> GT, nmaxpop)
1. PRINT 'WARNING: Tourn: list (nmax) exceeds local bound'
C CALL multi(nmax, LIST)
C set offspring (child) counter and initiate new rounds until new pop-
C -action complete (noff = nchild)
C noff = 0
C AN3 FORTRAN WHILE-DO (if labelled 101 with goto below)
C 101 IF (nleft.LT.nchild) THEN
C set competitor counter for current round then proceed until all parents
C for this round have competed. In this round or child population complete
C (may happen in last round)
C nleft = nmax
C CALL FORTRAN REPEAT-UNTIL (labelled 102 if for goto below)
C 102 CONTINUE
C set competitors group size and number of winners for current
C context (usually ncomp, nwin as specified; possibly at end of round)
C if within 2 groups of end then check
C IF (nleft.LT.(2*ncomp) .AND. (nleft.LT.2*ncomp)) THEN
C nleft = ncomp
C ELSE
C otherwise split into two contexts
C IF (nwin.GT.(nchild-noff)) THEN
C nleft = ncomp
C ELSE
C nleft = MIN(nleft/2,0.0)
C END IF
C then adjust number of wins to suit
C svic = 1 + INT(nleft/nwin/ncomp) - 1
C IF (nleft.LT.1) svic = 1
C ELSE
C nleft = ncomp
C END IF
C svic = nleft
C END IF
C randomly select group from parent pop that has not yet competed in this
C round - last group selects itself as reminder and list ready
C IF (nleft.GT.nleft) THEN
C CALL papart(nleft.nmax, nleft.LIST)
C ELSE
C all competed for stop condition
C nleft = nleft-nleft
CENDIF
C COMPETitors now in list (nleft=1) to (nleft-nleft) so use
C their fitnesses in pairs to pick n left offspring from nleft
C competitors in this context
C DO 105 j = 1, svic
C store first value as total for best
C ! top = (nleft + 2) / 2
C Do 105 j = (top), (nleft-nleft)
C IF (nleft.LT.top) THEN
C top = parity(list(top))
C END IF
C now sort by comparison with rest
C DO 105 j = (top), (nleft-nleft)
C IF (parity(list(j)) .GE. top) THEN
C top = parity(list(j))
C END IF
C 105 CONTINUE
C move best to top of range in list
C temp = list(nleft-1)
C list(nleft-1) = list(top)
C list(top) = temp
C create offspring and increment count
C noff = noff + 1
C DO 104 i = 1, nmax
C child(i,noff) = parent(1, list(nleft-1))
C 104 CONTINUE
C AN3 FORTRAN REPEAT-UNTIL (if goto 102 above)
C 1001 IF (nleft.LT.1) THEN
C
```c
        x-over in groups of across unless last group left is less
            across = across
            IF (n.left, 2), across < left
            CALL Map (x-over) gene, n.left, LIST
            IF x-over probable, use these pointers to identify chromosomes
                   and proceed with cross-over
            IF (Random().L.CROSS_SYSTEM) THEN
                Get n-points x-over locations in range 1 to (gene-1) and set up
                extra entry to terminate x-over at last gene
                gene = gene-1
                CALL Map (min. 1, n-point, CROSS)
                gene = gene(points) = gene
            Count through Icroms from start (i=1) in jumps of 2 (start+end)
                to get sections of chromosomes that are crossed-over (with
                intermediate parts unchanged)
                DO 1002 K = 2, across
                CONTINUE
            Finally x-over last (k=across) with stored chromosome
                pop(j), list(n.left=across) = store(k)
                CONTINUE
                1002
                CONTINUE
                1004
                END
            and increment count of those crossed
            k.cros = k.cros + across
            END
            440 ENDIF
            441 C ANSI FORTRAN WHILE-DO (goto 1001 above and endif)
            442 GOTO 1001
            443 END
            444 C End of subroutine XOVER
            445 C
            446 SUBROUTINE MUTATE(p, n, age, pop, KMUT, POP)
                INTEGER NGENE, NP, KMUT
                REAL KMUT, POP(p gene, ppop)
                C MUTATE a population for a GA
                C AA 62 vi 95 rev 26 vi 95
                451 C (c) Copyright Alexander Anderson 1995. All rights reserved.
                452 C NO WARRANTY.
                C
                C Modifies a population of chromosomes POP(p gene, ppop) where
                C NPOP is the population size and NGENE is the chromosome length
                C (ie number of genes) according to a chromosome mutation probability
                C KMUT (0 <= KMUT < 1). KMUT counts the mutations in this generation.
                C Modifies only POP and NOT other parameters or global
                C variables (except through function RANDOM)
                C
                C Uses function FOLLOW and subroutine MUTCHER
                C
                C
                INTGENE = 0
                C
                C Start
                C Initialise mutation counter for this generation
                KMUT = 0
                452 C For each chromosome (ppop of them) if random mutation occurs
                C then mutate that chromosome, otherwise unchanged (no action)
                C and increment mutation count
                DO 451 K = 1, npop
                IF (Random() .LE. KMUT) THEN
                    C Mutate this chromosome (pass column of POP as vector)
                    CALL Mutate (gene, POP(F,K))
                    KMUT = KMUT + 1
                ENDIF
                451
                C End of subroutine MUTATE
            C
            460 END
            C
            461 C SUBROUTINE MUTCHER(pgene, CHROM)
                C
                C MUTATE a chromosome POP(pgene) according to a chromosome mutation
                C probability and applying a case-specific mutation (through
                C subroutine MUTCHER) to it.
                C
                C Modifies only (kmute(pgene) and not the other parameter or global
                C variables (except through function RANDOM)
                C
                C Uses function FOLLOW and subroutine MUTCHER (letter to allow
                C case-specific mutation of a gene.
                C
                REAL KMUT
                INTEGER PGENE
                C IGENE is random index of gene picked for mutation
                IGENE = Random()
                C Start
                C Randomly select gene index for mutation in range 1 to n gene
                C gene = IGENE(RAND(0,PGENE-1))
                C Mutate this single gene - subroutine MUTCHER allows for
                C case-specific gene mutation operation
                C CALL Mutate (POP(IGENE))
                RETURN
                C}
```
Appendix A

Page 17

Aug 30 1995 11:18:31

C End of subroutine Invert
C
C SUBROUTINE INVERT (gene, CHROM)
C
C INVERT GENES
C
C REAL CHROM(gene)
C
C INV = bit a CHROM-verse
C
C AA 26 V1.95 rev 10 V1.95
C
C (c) Copyright Alexander Anderson 1995. All rights reserved.
C
C No Warranty.
C
C Given a CHROM(gene) of length n GENES genes carries
C out inversion (overwrites) chromosome by randomly selecting two
C gene locations between (and including) 1 and n gene and reversing
C order of genes in that section of the chromosome (rest unchanged).
C
C Modifies only CHROM(gene) and not other parameter or global
C variables.
C
C REAL RANDOM
C
C Uses function RANDOM
C
C REAL QTMP
C
C QTMP is temporary gene storage for exchange
C
C INTEGER ISTART, IEND, IINDEX, IEXK, IEXL, ILEFT, IRIGHT, I
C
C ISTART and IEND are the randomly selected start
C and end locations for inversion
C
C IINDEX is the number of genes to be inverted
C
C IEXK and IEXL are the starting locations for swaps
C
C ILEFT and IRIGHT are the indices of the pair swapped
C
C I is loop counter
C
C Start
C
C Randomly start and end gene locations in the range 1 to n gene
C Start location first
C ISTART = I+QTMP(Random(1,n))
C
C and end location different from the start
C IINDEX = I+QTMP(Random(1,n))
C
C ANS: PORTRAN REPEAT-UNTIL (labelled 2301 if 1-goto just below)
C
C If (IEND.EQ.ISTART) GOTO 2301
C
C IEND = I+QTMP(Random(1,n))
C
C Get total number of genes to be inverted (could be first)
C ILEFT = ISTART-1
C
C If (ILEFT.NE.1) ILEFT = ILEFT-1
C
C Do the required number of gene pair exchanges
C DO 2301 I = 1, ITMP
C
C Get locations of pair to be exchanged with wrap-around
C ILEFT = ILEFT-1
C IF (ILEFT.LT.1) ILEFT = (ILEFT-1) mod n
data a
C
C IRIGHT = ISTART+I-1
C IF (IRIGHT.GT.n) IRIGHT = (IRIGHT-1) mod n
data b
C
C Then exchange genes between this pair
C GTEMP = CHROM(ILEFT)
C CHROM(ILEFT) = CHROM(IRIGHT)
C CHROM(IRIGHT) = GTEMP
C
C 2302 CONTINUE
C RETURN
C
C End of subroutine Invert
C
C SUBROUTINE UNIQUE (gene, npop, KALTER, POP)
C
C INVERT GENES
C
C REAL POP(gene, npop)
C
C Ensures all members of a population are UNIQUE by making twins
C then an end gene and reversing the order of genes in that section
C (within a gene tolerance gene declared below) and false if any one of
C the genes does not satisfy this
C
C Given a population POP(gene, npop) of NP0P chromosomes each of
C
Page 18

Aug 30 1995 11:18:31

C GENES genes, checks if any are duplicates and if so makes these
C (to avoid running simulation redundantly and to ensure diversity)
C counting the number of such mutations in KALTER (which is assumed
C to have been initialised before subroutine call, is not initialised
C here). If the mutation is random there is no absolute guarantee
C that a single cell will ensure zero duplications, though depending
C on the case-specific mutation operator this is probable.
C
C Modifies only KALTER and POP (later by overwrite) and not other
C parameters or global variables.
C
C Uses function CIRAND and subroutine MIRCH.
C
C LOGICAL CHROM, SAME
C
C SAME is true for two chromosomes identical
C
C CHROM is index of chromosome being compared with rest
C ITMP is index counter for rest in comparison
C
C Start
C
C Initialise chromosome counter to go through entire population
C ITMP = 0
C
C Start with first chromosome then next, etc.
C ANS: PORTRAN REPEAT-UNTIL (labelled 2302 for 1-goto below)
C
C 2301 ITMP = ITMP+1
C
C Initialise comparison counter with next chromosome (others
C checked in previous rounds)
C ICOMP = ICOMP-1
C
C Compare with all others that have not been through process
C ANS: PORTRAN REPEAT-UNTIL (labelled 2202 for 1-goto below)
C
C 2202 ICOMP = ICOMP+1
C
C Check to see if any pass columns of pop as vectors
C same = CHROM(gene, npop, ICOMP, ICOMP, ICOMP, ICOMP)
C
C If the same, mutate chromosome (not comparison in case
C of triplets) (pass column of pop as chromosome vector
C and increment change counter parameter
C IF (same.NE.CHROM) THEN
C CALL MIRCH(gene, npop, ICOMP, ICOMP, ICOMP, ICOMP)
C
C CALL INVERT(gene, npop, ICOMP, ICOMP, ICOMP, ICOMP)
C possible alternative
C (invert instead)
C bit change record to save(1 not 4) when
C calling from Genes
C CALL NEUCH(gene, npop, ICOMP, ICOMP, ICOMP, ICOMP)
C possible alternative
C (new instead)
C kaler = kaler+1
C END ITMP
C
C Terminate comparisons if same found or all completed
C ANS: PORTRAN REPEAT-UNTIL (1-goto 2202 above)
C IF (ITMP.EQ.NP0P) THEN
C ANS: PORTRAN REPEAT-UNTIL (1-goto 2202 above)
C IF (ICHROM.LT.ICOMP) GOTO 2202
C
C Terminate process when last two compared with each other
C ANS: PORTRAN REPEAT-UNTIL (1-goto 2202 above)
C IF (ICHROM.LT.(ITMP-1)) GOTO 2301
C
C RETURN
C END
C
C End of subroutine Unique
C
C FUNCTION CIRAND(gene, chrom, chron2)
C
C LOGICAL CHROM
C
C INTEGER CHRON, KALTGB
C
C REAL CHROM(gene), CHRON(gene)
C
C True if two CHROMosomes are the SAME
C
C AA 26 V1.95
C
C (c) Copyright Alexander Anderson 1995. All rights reserved.
C
C No Warranty.
C
C Compares two chromosomes CHROM(gene) and CHRON(gene) both
C of length n GENES genes, gene by gene and returns true if they
C are identical (within a gene tolerance gene declared below)
C and false if any one of the genes does not satisfy this.
C
C Equivalent to:
C
C (c) Copyright Alexander Anderson 1995. All rights reserved.
C
C No Warranty.
C
C Given a population POP(gene, npop) of NP0P chromosomes each of
C INTEGER 1
C I is gene index counter
C LOGICAL SAME
C SAME true while corresponding genes the same
C Start
C Initialize gene index counter and assume same is true
C
1 = 0
C 1 = .TRUE.
C Go through all genes until all done or one different
C ANSI PORTTRAN REPEAT-UNTIL (labeled 2101 with if-goto below)
2101 IF (I .NE. 1) THEN
C Compare corresponding genes
C IF (ABS(A(ITEM(I)) - A(ITEM(J))) .GT. 0.001) SAME .FALSE.
C TERMINATE as soon as different or all genes compared
C ANSI PORTTRAN REPEAT-UNTIL (if-goto 2101 above)
C IF (SAME .EQ. .TRUE.) AND (.NOT..I.LT.NGENE) GOTO 2101
C SAME = same
C RETURN
C END
C End of function Chame
C
C SUBROUTINE RESULT(IPOS, MAXGEN, NGENE, NCREAM, FCREAM, TCREAM,
C ICREAM(I), REVOL(I,MAXGEN), CEVO(I,MAXGEN),
C REAL FCREAM(TCREAM, NCREAM), FCREAM(NCREAM), REVOL(I, MAXGEN),
C CEVO(I,MAXGEN), NCREAM, TCREAM
C Prints final RESULTS
C AA 21 vi 95 rev 26 vi 95
C (c) Copyright Alexander Anderson 1995. All rights reserved.
C No Warranty.
C
C After IGEN out of maximum MAXGEN generations with NGENE genes in
C each chromosome, prints out NCREAM best chromosomes CCREAM(ncore)
C (ncore) with fitnesses FCREAM(ncore) using rank pointers LCREAM;
C ncore and evolution record in IREVO(1,MAXGEN), REVOL(1,MAXGEN)
C and CEVO(I,MAXGEN) (defined in subroutine Genalg).
C Does not modify any parameters or global variables.
C
1 = 0
C I = 0
C J = 0
C Start
C PRINT *, 'EVOLUTION PROCESS OVER', IGEN, '<>', NGENE
C PRINT *, ' GENERATIONS'
C DO 1001 I = 1, IGEN
C PRINT *, 'GENERATION', I
C PRINT *, '
C PRINT *, ' Popl No No- No-Cro-No.of No.of No.better',
C PRINT *, ' No.of',
C PRINT *, ' Size placed sizeOver Mutats Inverses than prov',
C PRINT *, ' Simulations'
C PRINT *, 'Fitness Fitness Fitness Gen P devdation',
C PRINT *, ' Best chromosome of generation:',
C PRINT *, ' (noream(J), I, J, NGENE)
1001 PRINT *, ' 
1001 CONTINUE
C RETURN
C End of subroutine Result
C
C SUBROUTINE PHIKEP (ipple, a Ritch, NL, NLIST, LISt)
C INTEGER NPICK, NLST, NL, LISt(NLIST)
C Randomly P-1 enk pointers from an integer array and partition it
C AA 08 vi 95
C (c) Copyright Alexander Anderson 1995. All rights reserved.
C No Warranty.
C
C Randomly select PICK pointers from elements L to NLST of integer
C array LIST (n list with NLST entries, and then move them by exchange
C to locations (left-pick) to NLST, finally decrementing
C NLSTP (left-pick) so that elements yet to be chosen are in
C locations 1 to NLST and chosen ones are in (left+pick) down to
C 1. Set NLSTP = pick. Assumes prior checks for nlst < leftpick.
C Modifies only NLST and order of elements in LIST and not other
C parameters or global variables (except through function RANDOM).
C
C REAL RANDOM
C Use function RANDOM.
C INTEGER 2, 1, 1, 1
C 1 = loop counter
C 2 = RANDOM is randomly selected index
C 1 = zero temporary storage for element exchange
C Skips
C Do this for each of Npick elements
C DO 901 1 = 1, NPICK
C Set random pointer in range of n 리스트 to (left+pick) to NLST
C 1 = RANDOM(Random(1) + pick)
C then move chosen element to (left+pick) with exchange
C elements moving above to remain available for selection
C iftemp = list(temp)
C list(temp) = list(temp+pick)
C compile
C DO 901 1 = 1, NPICK
C RETURN
C End
C End of subroutine Phikep
C
C SUBROUTINE MALLIST (nlist, LIST)
C INTEGER MLIST, LIST(NLIST)
C M-make a LIST of randomly generated integer pointers
C AA 02 vi 95
C (c) Copyright Alexander Anderson 1995. All rights reserved.
C No Warranty.
C C Fills integer array LIST(nlist) with pointer values in range
C 1 to NLIST in a random order.
C Modifies only LIST and not other parameters or global variables
C (except through function RANDOM).
C REAL RANDOM
C Use subroutine MAXSEQ and function RANDOM
C
DATA MIN, INCR /1, 1/
POINTER start at MIN in increments INCR = 1
INTEGER 1, JARRAY, ITEMP, min, incr
I is loop counter
JARRAY is randomly selected pointer
ITEMP is temporary storage for an exchange

Start:
Initialize list in simple sequence from random start
CALL NWMARK(min, incr, nlist, LIST)
Randomly re-order list until only last entry is left
DO 301 = 1, (nlist-1)
   Pick random pointer in range i to nlist
   jran = I*INT(RAND) + (nlist-1)
   Swap list(i) and list(jran) so latter fixed
   iTEMP = list(i)
   list(i) = list(jran)
   list(jran) = iTEMP
   Return from (1-1) to nlist yet to be done
   C 301 CONTINUE
   RETURN
C End of subroutine Manlist

C SUBROUTINE NWMARK(min, max, nmark, MARKER)
INTEGER MIN, MAX, NMARK, MARKER
H-As a MARKER list ordered randomly selected pointers
C AA 06 v2 93
(c)Copyright Alexander Anderson 1995. All rights reserved.
C No Warranty.
C Fill first NMARK elements of integer vector MARKER with ordered
C (low first/high last) pointers randomly but exclusively chosen
C within range MIN to MAX (assumes prior checks for 1 <= min <= max
C and (max-min) >= nmark)
C Modifies only order of MARKER and not other parameters or global
C variables (except through function RAND)
REAL RANDOM
C Uses subroutines MSEQL and ISORT and function RANDOM
INTEGER INCR
DATA incr /L,
   pointers go up in increments INCR = 1
INTEGER 1, JARRAY, MARKER, ITEMP
I is a counter
JARRAY is a randomly selected pointer to marker
MARKER is (max-min) for convenience
ITEMP is temporary storage for exchange

Start:
Calculate a data range for convenience
maxrange = (max-min)
set up a sequential list from min to max with random start point
CALL NWSEQL(min, incr, maxrange, MARKER)
Then randomly pick maxnumber of these and put into first maxnumber locations
DO 601 = 1, maxnumber
   jran = I*INT(RAND) + (maxrange-1)
   iTEMP = marker(jran)
   marker(jran) = marker(i)
   marker(i) = iTEMP
   601 CONTINUE
C Finally sort these first maxnumber elements into ascending order
CALL ISORT(maxrange, MARKER)
RETURN
C End of subroutine MNWSEQL

C H-As a SEQUENTIAL L-list with a random start and wrap-around
ENDIF
C
492 CONTINUE
RETURN
END
C END of subroutine Insert
C
C SUBROUTINE TRANS(ival, rval, POINT)
INTEGER NVAL, POINT(nval)
REAL RVAL(nval)
C get 1st toper rank 2-index vector for elements of a real vector
FA 08 vili 95
C No warranty.
C
C Non-recursive Quickstart on one-dimensional real array (vector).
C RVAL(nval) with NVVAL real elements to produce one-dimensional
C linear array POINT(nval) the elements of which are rank pointers
C (ie, exclusive integers in range 1 to nval) such that RVAL(POINT(i))
C values are in ascending order for i=1,2,...,nval (without re-ordering)
C rval(nval).
C
C References: EM Reingold and DH Reingold, Pacalgorithms. An
C introduction to programming, Scott Foreman/Brown Little,
C Glenview Ill, 1986, Sect 8.4
M Pramesh et al, Numerical Recipes in FORTRAN, 2nd edn,
C Cambridge UP, Cambridge, 1992, Chap 8
C Modified only POINT(nval) (self-initialising) and not any other
C parameters or global variables.
C
INTEGER NVMD, NSTACK PARAMETER (nstack = 265)
DATA nsort /7/
C
C Below NSORT >= 3 (minimum possible value from Reingold); remainung elements in
C subarray sorted by insertion sort (not
C Recipes). Without recursion (eg Reingold) in Fortran need auxiliary
C push-down stack with up to NSTACK slots (eg Recipes) - set
C nsort=nstack to 2 if pnum number of elements) to be safe (eg 185
C from modern maxp=1000) - code has WARNING if exceeded
C
INTEGER ISTACK(nstack), JSTACK, LEFT, RIGHT, MED, ITEM, IHOLE, l,
J, L, 0
ISTACK(nstack) is stack and JSTACK is stack counter
LEFT and RIGHT are lefthead and rightmost element indices
for current subarray in Quicksort (Initially 1 and nval)
MED is median between left and right indices of subarray
ITEM is temporary storage for exchange (Quickstart) or
'HOLE' index (insertion sort)
IHOLE is 'hole' point (value index item) for insertion sort
I and J are counters (in Quickstart 1 moving right and J
moving left)

C Start
C
C Before sort initialise point(nval) so that it can be re-ordered
C DO 801 I = 1, nval
C IF (point(I)) = 1 THEN
C 801 CONTINUE
C
C and insert in place, continuing with next to right up to (right)
C DO 804 J = (left+1), right
C Insert point(J) into place amongst point(I) to J-1
C - index of first one for comparison
C I = J-1
C and initialise a 'hole in point' index memory in case its index overwritten when shifting elements up
C item = left
C hole = point(I)
C ANI FORTRAN REPEAT-UNTIL (label 803 with if-goto below)
C CONTINUE
C IF (rval(hole).LT.rval(point(I))) THEN
C Shift the 'hole' down
C point(I+1) = point(I)
C I = I + 1
C ELSE
C ITEM
C Remove where 'hole' is and stop
C item = I + 1
C I = 1
C ENDIF
C ANI FORTRAN REPEAT-UNTIL (if-goto 803 above)
C IF (ITEM.GT.0) GOTO 803
C Insert pointer into right place
C point(ITEM) = hole
C CONTINUE
C After insertion sort strip next subarray indices off stack
C - unless this is last sort (then no more left to do)
C JSTACK = JSTACK-2
C IF (JSTACK.GT.0) THEN
C ITEM = point(JSTACK-2)
C JSTACK = JSTACK-1
C ENDIF
C ELSE
C Sufficient in subarray = Quickstart these and push subarray
C indices onto stack
C
C Get sentinel and partitioning elements - first get median
C item = (left+right)/2
C Use left as partitioning element and shift median to
C (left-1) as sentinel (right is other)
C item = point(left)
C point(left) = point(right)
C point(right) = item
C TERNARY input: 3 comparisons to ensure sentinels large (right) and small
C (left-1) and partitioning element (left) between these
C IF (rval(point(left-1)).GT.rval(point(right))) THEN
C item = point(left-1)
C point(left) = point(right)
C point(right) = item
C ENDIF
C IF (rval(point(left)).GT.rval(point(right))) THEN
C item = point(left)
C point(left) = point(right)
C point(right) = item
C ENDIF
C IF (rval(point(left)).LT.rval(point(right))) THEN
C item = point(left)
C point(left) = point(right)
C point(right) = item
C ELSE
C TERNARY
C IF (ITEM.GT.0) GOTO 803
C Insert pointer into right place
C point(ITEM) = item
C CONTINUE
C
C Set scanning pointers for subarray to go right-ways from
C (left-1) and left-ways from right
C L = (left-1)
C J = right
C Do comparisons with partitioning element (left) until scanning
C pointers have crossed (locates where partition goes)
C ANI FORTRAN REPEAT-UNTIL (label 805 for if-goto below)
C CONTINUE
C Right scan from (left+1) for element > partition
C C ANI FORTRAN REPEAT-UNTIL (label 806 for if-goto below)
C CONTINUE
C 1 = 1-1
IF (IVAL (Point(1)), I.RVAL (Point (Point (Left))) ) GOTO 806
807
CONTINUE

IF (IVAL (Point (1)), I.RVAL (Point (Point (Left))) ) GOTO 807

Exchange high element to left and low to right of part’n.

IF (I.GE.1) THEN
  Itemp = point (1)
  point (1) = point (j)
  point (j) = Itemp
END IF

IF (IVAL (Point (1)), I.RVAL (Point (Point (Left))) ) GOTO 807

Partitioning of subarray complete when 3rd insert partitioning element point (Left) in right place by exchange.

Itemp = point (Left)
point (Left) = point (j)
point (j) = Itemp

After Quicksort push pointers to larger subarray onto stack and continue processing smaller subarray (to keep stack as short as possible) - but warn if declared stack bound exceeded.

(Both ideas from Recipies)

stack = stack+2

IF (jstack GT stack)
  1
  PRINT 'WARNING: Stack about to overwrite bound'
  IF (I.EQ.1 .OR. I.EQ.1) THEN
    Right subarray is larger.
    jstack (jstack) = j
    jstack (jstack) = (j-1)
  ELSE
    Left subarray is larger.
    jstack (jstack) = (j-1)
    jstack (jstack) = i
    left = 1
  END IF
ENDIF

IF (IVAL (Point (1)), I.RVAL (Point (Point (Left))) ) GOTO 807

At this stage point (i)val completely so return
RETURN

End of subroutine Isrank

FUNCTION ISRAND

REAL RANDOM

return RANDOM number (uniform random deviate between 0.0 and 1.0)

AA 3 will 95

No Warrantee.


UP, 1992, pp273-4

but in entirely floating point form to reduce machine-dependency.

and without parameter or local save. NB: call must be in form:

Random (i-1) must have empty parentheses in standard FORTRAN.

INTEGER IICALL

COMMON IRAND/ IICALL, knuth

To avoid extreme parameter passing and to ensure that values are

saved from call to call then top-level block must declare this

common block IRAND and initialize IICALL = 0 before first call.

IICALL is counter to the KNUTH table which produces the randoms by

subtracting.

NB: the integers 21, 31, 53 and 83 (or 54) and important.

not be altered in the code.

REAL SEED, START

DATA SEED, start / 0.41749, 0.0000000003, /

SEED and START are initializing values - seed must be in the
Appendix A

Appendix A

50

Page 27

Aug 30 1995 11:18:31

C exclusively from range 1 to npop).
C
C REMAINDER IS FOR ILLUSTRATION PURPOSES ONLY
C Goldberg &cc(x) example (using n gene bits for positive integer
C where real genes set to 0.0 or 1.0 to mimic binary number bits
C and gene is high bit to low bit in order)
C
REAL X
C Temporary storage of n gene bit number for convenience
C INTEGER i, j, k
C J temporary storage
C Start
C For population evaluate each member in current run
DO 1302 k = 1, n
C Identify those to be simulated from pointer vector
i = i run(k)
C For each chromosome evaluate a fitness as an n gene -bit
C positive binary integer (high bit first in chromosome)
x = 0.0
DO 1301 j = 1, n+1
C IF(i pop(j, i) / 0.0 + 0.5) IF(i pop(j, i) / 0.0 + 0.5)
C x = x * 2**((gene - 1)
1301 CONTINUE
C Return x
C 1302 CONTINUE
C End of subroutine Simula
C
C SUBROUTINE NEWCHR(nGene, nChrom)
INTEGER nGene
REAL nChrom(nGene)
C Create a new chromosome (case-specific for model of binary number
C AA 27 01 95
C No Warranty.
C Creates a new chromosome with nGene real genes, 0.0 < gene < 1.0
C by a random process.
C Modifies only CHROM(nGene) and not other parameters or global variables
C (except through function RANDOM)
REAL RANDOM
C Uses function RANDOM
C REMAINDER IS FOR ILLUSTRATION PURPOSES ONLY
C This version sets genes to either 1.0 or 0.0 (models binary string)
C
INTEGER i
C I is loop counter
C For each of n genes in turn randomly set a value of 0.0 or 1.0
DO 1101 i = 1, n
C 2P Random (1, 0.5, 0.5) THEN
C CHROM(i) = 1.0
C ELSE
C CHROM(i) = 0.0
1101 CONTINUE
C End of subroutine Newchr
C
C SUBROUTINE MUTGEN(Gene)
REAL Gene
C MUT - rate Gene - (case-specific operator for model of binary number
C AA 27 01 95
C No Warranty.
C
APPENDIX B. MODULE DESCRIPTIONS AND FLOW CHARTS

This contains module descriptions and also flowcharts where module complexity warrants it. Table B.1 provides an index to subprograms for Appendices A and B.

B.1 Principal GA subroutine GENALG

Subroutine GENALG (npar):

(a) establishes the GA options and population characteristics, principally through editing of its declarations (see Section 5.4 and 5.5);

(b) carries out the basic "simple" generational GA (outlined in Section 3.1), calling lower-level modules (described elsewhere in Appendix B) for most of the basic GA operations; and

(c) records and outputs details of the generational evolution process Section 2.5).

It must be called from some other software or main program which establishes the value of a single parameter NPAR - the initial parent population size (ie number of chromosomes).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>npar</td>
<td>input</td>
<td>Integer value of initial population size. No check is made that: npar &lt;= maxpop (see PARAMeter declarations for maxpop)</td>
</tr>
</tbody>
</table>
Tests indicate convergence?

End process

Reproduction will occur

Get next (child) generation population size

Function NEXTPOP (Appendix B.9)

Selection (Section 3.2)

Get next (child) generation population pool (including any change in population size) by tournament selection

Subroutine TOURN (Appendix B.6)

Reproduction (Section 3.1)

Subroutine XOVER (Appendix B.7)

Apply cross-over operator to whole child population

Apply mutation operator to whole child population

Subroutine MUTATE (Appendix B.5)

Apply inversion operator to whole child population

Subroutine INVERT (Appendix B.6)

Diversity (Section 4.1)

Subroutine UNIQUE (Appendix B.3)

Eliminate child population duplicates by additional mutations

Set while termination flag

Transpose child population into parent population ready for next generation

endif

endwhile
Output stored results when GA complete

return

Subroutine RESULT
(Appendix B.9)
<table>
<thead>
<tr>
<th>Name</th>
<th>type</th>
<th>see App. B</th>
<th>Listing in App. A</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHSAME</td>
<td>function</td>
<td>B.11</td>
<td>2o</td>
</tr>
<tr>
<td>CREAM</td>
<td>subroutine</td>
<td>B.5</td>
<td>2d</td>
</tr>
<tr>
<td>GENALG</td>
<td>subroutine</td>
<td>B.1</td>
<td>1</td>
</tr>
<tr>
<td>INIPOP</td>
<td>subroutine</td>
<td>B.3</td>
<td>2a</td>
</tr>
<tr>
<td>INVCHR</td>
<td>subroutine</td>
<td>B.8</td>
<td>2m</td>
</tr>
<tr>
<td>INVERT</td>
<td>subroutine</td>
<td>B.8</td>
<td>2l</td>
</tr>
<tr>
<td>IRANKP</td>
<td>subroutine</td>
<td>B.11</td>
<td>3f</td>
</tr>
<tr>
<td>ISORTI</td>
<td>subroutine</td>
<td>B.11</td>
<td>3e</td>
</tr>
<tr>
<td>MKLIST</td>
<td>subroutine</td>
<td>B.10</td>
<td>3b</td>
</tr>
<tr>
<td>MKMARK</td>
<td>subroutine</td>
<td>B.10</td>
<td>3c</td>
</tr>
<tr>
<td>MKSEQL</td>
<td>subroutine</td>
<td>B.10</td>
<td>3d</td>
</tr>
<tr>
<td>MUTATE</td>
<td>subroutine</td>
<td>B.8</td>
<td>2j</td>
</tr>
<tr>
<td>MUTCHR</td>
<td>subroutine</td>
<td>B.8</td>
<td>2k</td>
</tr>
<tr>
<td>MUTGEN</td>
<td>subroutine</td>
<td>B.2</td>
<td>4c</td>
</tr>
<tr>
<td>NEWCHR</td>
<td>subroutine</td>
<td>B.2</td>
<td>4b</td>
</tr>
<tr>
<td>NEWPOP</td>
<td>function</td>
<td>B.6</td>
<td>2g</td>
</tr>
<tr>
<td>PKPART</td>
<td>subroutine</td>
<td>B.10</td>
<td>3a</td>
</tr>
<tr>
<td>POPOUT</td>
<td>subroutine</td>
<td>B.9</td>
<td>2f</td>
</tr>
<tr>
<td>POSTAT</td>
<td>subroutine</td>
<td>B.5</td>
<td>2e</td>
</tr>
<tr>
<td>RANDOM</td>
<td>subroutine</td>
<td>B.11</td>
<td>3g</td>
</tr>
<tr>
<td>REFRSH</td>
<td>subroutine</td>
<td>B.4</td>
<td>2b</td>
</tr>
<tr>
<td>Name</td>
<td>type</td>
<td>see App. B</td>
<td>Listing in App. A</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>RESULT</td>
<td>subroutine</td>
<td>B.9</td>
<td>2p</td>
</tr>
<tr>
<td>RNSIMR</td>
<td>subroutine</td>
<td>B.4</td>
<td>2c</td>
</tr>
<tr>
<td>SIMULA</td>
<td>subroutine</td>
<td>B.2</td>
<td>4a</td>
</tr>
<tr>
<td>TOURN</td>
<td>subroutine</td>
<td>B.6</td>
<td>2h</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>subroutine</td>
<td>B.3</td>
<td>2n</td>
</tr>
<tr>
<td>XOVER</td>
<td>subroutine</td>
<td>B.7</td>
<td>2i</td>
</tr>
</tbody>
</table>
B.2 User-defined interface subroutines SIMULA, NEWCHR and MUTGEN

The GA optimisation process is essentially independent of the problem being optimised (Goldberg 1989) and therefore the GA code should be essentially independent of the problem simulation software. However:

- the chromosomes must be “coded” to represent the problem being optimised and in this way the gene sequences can be built into meaningful chromosomes. The chromosomes are “decoded” to provide data to drive the simulation software, with the output from the simulation interpreted in the form of fitnesses which drive the GA; while

- the nature of the problem and its coding directly affects the nature of the genes (and hence the gene operation of mutation).

Thus three “interface” routines are necessary to provide the connection between the GA software described in this report and the user-defined code/software required for the above three functions. These “interface” calls are defined in Appendix A simply to provide the basic parameter-passing necessary to make the GA operate. They may well require other information to be passed relating to the specific application (e.g. through COMMON blocks) and the user must code for this.

(a) subroutine SIMULA (ngene, npop, lrun, pop, POPFIT):

The decoding/simulation/fitness evaluation interface requires the following information which is conveyed by the parameters set up:

The population of chromosomes consists of the NPOP columns (each with NGENE genes) of the array POP (ngene, npop) and these must be decoded by user defined code to provide input data to the simulation software;

The simulation will be carried out for NRUN (1<=nrun<=npop) of the member chromosomes, being those identified by the first nrun elements of the integer pointer vector LRUN (npop); and

The results of the simulation for these chromosomes will be coded into fitnesses POPFIT (npop), indexed in the same order as POP, with these being returned to the GA to influence its operations.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ngene</td>
<td>input</td>
<td>Integer chromosome length (i.e. number of genes) for dynamic array sizing of pop.</td>
</tr>
<tr>
<td>npop</td>
<td>input</td>
<td>Integer population size (i.e. number of chromosomes) for dynamic array sizing of $\ell$ run, pop and popfit.</td>
</tr>
<tr>
<td>nrun</td>
<td>input</td>
<td>Integer number of chromosomes to be simulated this call, i.e. those identified by first nrun pointers in $\ell$ run.</td>
</tr>
<tr>
<td>$\ell$ run</td>
<td>input</td>
<td>Integer pointer vector (Section 5.3) whose first nrun members identify chromosomes in pop to be simulated this call.</td>
</tr>
<tr>
<td>pop</td>
<td>input</td>
<td>Real two-dimension array of current population (npop chromosomes each with ngen genes). The nrun chromosomes identified by the first nrun pointers in $\ell$ run will be &quot;decoded&quot; to provide data for the simulation.</td>
</tr>
<tr>
<td>POPFIT</td>
<td>output</td>
<td>Real vector of fitnesses corresponding to chromosomes. Each simulation must be &quot;encoded&quot; to provide a fitness value for the corresponding chromosome.</td>
</tr>
</tbody>
</table>

(b) subroutine NEWCHR (ngen, CHROM):

Whenever a new chromosome is generated (e.g. for the initial "seed" population, for an influx of "incomers" to a population or to replace an infeasible or duplicate chromosome) then the nature of the problem and its coding will influence what that chromosome and its genes may look like. Eg. in the example of Section 5 and Appendix C, genes may only take the values 0.0 or 1.0 with equal probabilities. Thus user-defined code is necessary for a new chromosome CHROM (ngen) with NGENE genes. Subroutine NEWCHR has the same parameter list as subroutines MUTCHR and INVCHR:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| ngene    | input   | Integer chromosome length (i.e. number of genes) for dynamic array sizing of chrom.  
| CHROM    | output  | Real vector chromosome of ngene genes which should be randomly generated (in range 0.0 to 1.0 in subroutine NewChr (Section 5.4) |

(c) subroutine MUTGEN (GENE):

The reproduction operators of cross-over and inversion simply redistribute genes within the population or a chromosome, respectively, but mutation changes the genes and consequently may depend on the nature of the problem and its coding. In addition, a variety of different mutation operators have been described (“bitwise”, “creeping (or adjacency)”, etc). Thus user-defined code is necessary for a new gene GENE. Eg, in the example of Section 5 and Appendix C, genes simply exchange the value 0.0 for 1.0 and vice versa.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENE</td>
<td>input/output</td>
<td>Real gene (range 0.0 to 1.0) whose value will be changed by code in subroutine MutChr (Section 5.4).</td>
</tr>
</tbody>
</table>

B.3. Eliminating duplicates in the populations - subroutines INIPOP and UNIQUE

The first population to be simulated is a randomly generated initial “seed” population. All subsequent populations to be simulated are the products of selection and reproduction. In both cases, though, duplicate chromosomes are eliminated to avoid the unnecessary expense of repeating identical simulations and also to ensure genetic diversity in the populations. In both cases, only the chromosomes are available and not their fitnesses (the simulation for these has not yet been done).

A. subroutine INIPOP (npar, ngene, ncream, fitmin, PARENT, FCREAM, LCREAM)

This has three functions:
(i) it outputs this “seed” population (these lines could be commented out).

(ii) it randomly generates an initial parent population of NPAR chromosomes (each with NGENE genes) stored in the columns of PARENT (ngene, npar), using subroutine NEWCHR described in Appendix B.2 above, calling this until each new chromosome is different from those already generated; and

(iii) it initialises the store of the NCREAM best fitnesses (see Appendix B.5 below) found, FCREAM (ncream), all as some minimum fitness FITMIN, and the integer vector LCREAM (ncream) which contains the rank pointers to these (it is not necessary to set up any chromosomes in CCREAM).

The first and last processes are straightforward. The middle one proceeds as in the flowchart on the next page.
Randomly generate the first chromosome PARENT(1,i) for i=1

Subroutine NEWCHR.
(Appendix B.2)

For remainder of seed population
For i:=2, npar, +1 do

Randomly generate another chromosome PARENT(1,i)

Subroutine NEWCHR
(Appendix B.2)

Compare with those previously found, i.e. j=1 to (i-1), and replace if and until different to these

j := 1

repeat

label 1800

PARENT(1,j) = PARENT(1,i)?

T Function CHSADE
(Appendix B.11)

F

Randomly generate a replacement PARENT(1,i)

endif and start again
j := 0

increment comparison counter
j := j+1

until j = i

label 1801

endo
B. subroutine UNIQUE (ngene, npop, KALTER, POP):

This takes population of NPOP chromosomes (each with NGENE genes) passed as columns of POP (ngene, npop) and compares them all. If a duplicate is found, then a new chromosome is randomly generated, with KALTER counting the number of these occurrences.

The difference between INIPOP and UNIQUE is that with the latter there is already a complete population, whereas the former creates a population one chromosome at a time. INIPOP checks each new member against all existing ones, thus guaranteeing uniqueness, whereas UNIQUE only changes a member if another similar one if found but, because the change is random and it does not restart the entire process, cannot absolutely guarantee uniqueness.
enter

Initialise chromosome counter
ichrom:=0

label 2201
repeat

Start with first chromosome, then next, etc until all done
ichrom:=ichrom+1

Initialise comparison counter for next chromosome
(previous ones all different from previous rounds)
icomp:=ichrom

Compare with all others that have not been through process

label 2202
repeat

Look at next, then next, etc
icomp:=icomp+1

Are they the same?
POP(1,ichrom) = POP(1,icomp)

Function CHSAME
(Appendix B.11)

T

F

NEWCHR

Randomly generate a replacement
POP(1,ichrom)

Increment changes counter
kalter:=kalter+1

endif

Until two the same OR all done

F

T

Until all checked (except last which will be unique by previous round)
ichrom=(npop-1)

return
B.4. Simulations with population management - subroutines REFRSH and RNSIMR

The simulations which create the fitnesses that “drive” the GA are created by three subroutines:

(i) subroutine REFRSH (nfresh, pfresh, ffresh, pfeasi, feasib, IFRESH, PREPL, FITMIN)

This simply looks at the data for the population management which come from the declarations in subroutine GENALG (Section 4) and set up three general parameters IFRESH, PREPL and FITMIN which will be used in subroutine RNSIMR below to control the actual process:

<table>
<thead>
<tr>
<th>“Refresh” population with “incomers” every NFRESH generations</th>
<th>Normal operation or replace infeasible chromosomes with fitness &lt; FEASIB at any generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>If counter reaches NFRESH then reset it to zero to start counting again:</td>
<td>If counter has not reached NFRESH (set &gt; maxgen to avoid left) then continue incrementing:</td>
</tr>
<tr>
<td>IFRESH = 0</td>
<td>IFRESH = IFRESH + 1</td>
</tr>
<tr>
<td>and set up probability and fitness for a population refresh at this generation:</td>
<td>and set up probability and fitness for a normal generation (or for replacement of infeasibles):</td>
</tr>
<tr>
<td>PREPL = PFRESH</td>
<td>PREPL = PFEASI</td>
</tr>
<tr>
<td>FITMIN = FFRESH</td>
<td>FITMIN = FEASIB</td>
</tr>
<tr>
<td>otherwise proceed as on the right</td>
<td></td>
</tr>
</tbody>
</table>

(ii) subroutine RNSIMR (fitmin, prepla, ngen, npop, KREPLA, KSIM, POP, POPFIT, LFITRK)
This actually manages the simulation process. KREPLA and KSIM are simply counters for the number of chromosome replacements (for either reason in (i) above) and for the number of simulations, respectively, this generation. It takes an initial population of NPOP chromosomes (each with NGENE genes) in the columns of POP (ngene, npop) and may replace some of these (as in (i) above) while obtaining fitnesses for all of them in POPFIT (npop) and producing an integer rank pointer array LFITRK (npop) for the fitnesses (see Section 2.3).

It is flow charted below and carries out the actual simulation via an “interface” by calling:

(iii) subroutine SIMULA - see Appendix B.2(a) above.
enter

Initialise replacement and simulation counters and set first pass flag
kritpa:=0 (zero is normal)
klim:=0 first:=true

Set up pointer vector to initially point to all members of population
(as yet without fitnesses)
For all i:=1, npop, do lfitrk(i):=i

Set up run counter to first simulate all population members
nrun:=npop

While run counter indicates some left to do, then keep doing them: nrun > 0?

T
Run simulation for nrun
(after first cycle the least fit) members as identified
by first nrun pointers in lfitrk (all of them the first
time) so as to get the fitnesses POFFIT(npop)

Subroutine SIMULA
(Appendix B.2)

F

Sort the pointers in lfitrk(npop) to index the fitnesses
popfit(npop) in ascending order

See Section 2.3
-Subroutine TRANKP
(Appendix B.11)

A

B

C

label 1201

label 1202

66
Scan the nold-nrun lowest fitnesses in ascending order to identify and replace candidates for replacement.

- Initialise nold:=nrun and nrun:=0 in case there are none left to do.

- For i:=1, nold, +1 do:
  
  - Compare with set fitness and probability (from REFRESH).

  - If in first pass only increment replacement counter - if first = true:
    kreplo:=kreplo+1

  - Increment run counter and get a new member: nrun:=nrun+1

  - Do nothing if none to replace

- Endif

- Enddo

- Set first pass flag to false now: first:=false

- Endwhile

- Return
B.5 Fitness statistics (subroutine POSTAT) and best chromosomes from all generations (subroutine CREAM)

Having generated a full set of fitnesses (with a rank pointer index) for a parent population, before proceeding to selection and reproduction (Appendix B.6 below), the population of fitness statistics can be evaluated and recorded and the best chromosomes (the “cream”) from all generations can be recorded. In principle, neither are essential (unless the former are used for a convergence criterion in subroutine GenAlg or for variable population in function NEWPOP - see Appendix B.6 below) and they could be commented out:

- subroutine POSTAT (npop, lprank, popfit, PSTAT):

This does not need the chromosomes, only the NPOP population fitnesses POPFIT (npop) and the corresponding integer fitness rank pointer LPRANK (npop) - see section 2.3. Various fitness statistics are selected or computed and stored in the short array PSTAT (see code in Appendix A for exact definitions).

- subroutine CREAM (ngene, npop, popfit, pop, lpopra, ncream, KCREAM, FCREAM, CCREAM, LCREAM):

For a user specified number NCREAM of best chromosomes (declared in subroutine GenAlg) this creates FCREAM (ncream) and CCREAM (ngene, ncream) which contain the ncream best fitnesses so far and the corresponding chromosomes (each of length NGENE genes as columns) respectively. In addition LCREAM (ncream) is the integer fitness rank pointer index for these as described in Section 2.3 (LCREAM and FCREAM but not CCREAM have to be initialised in subroutine INIPOP - see Appendix B.2). The number added to this list at each generation is counted in KCREAM. At each generation the population POP (ngene, npop) with corresponding fitnesses POPFIT (npop) and integer rank pointer index LPOPREA (npop) (see Section 2.3) are compared with the cream and if any are better they are added (by overwriting), with kcream being increased.
Initialise counters for additions to cream and number of current generation scanned and flag for process termination:
kcream:=0  kpop:=0  swap:=true

While swap=true do

T

Cannot make more than the smaller of ncream or npop swaps:
kcream>=ncream OR kpop>=npop

F

Set termination flag to stop:
swap:=false

T

Assume next candidate not in cream - set:
there:=false

F

Initialise cream index counter ready for least fit first:
i:=0

repeat

Try next fittest
i:=i+1

Fitness equality?
popfit(ipopra (npop-kpop)) =fcram(lcream(i))?

F

Then see if chromosomes match - if so then set a flag: there:=true

T

label 1401

label 1402

Section 2.3 for pointers

Function C0SSAME
(App B.11)

A  B  C  D  E
Until all checked
OR one found:
(i:=ncream)
AND (there=true)
F

If better and not already in cream, then swap by overwriting
there=false?
T

Swap fitnesses:
fcream(1cream(1+kcream))
:=popfit(1popra(npop=kpop))

Swap chromosomes
gene by gene

Increment cream counter
for one added
kcream:=kcream+1

Do nothing

Increment population counter
for another one checked
kpop:=kpop+1

If any swaps done (if kcream>0)
re-index rank pointer for cream

See Section 2.3
-Subroutine IRANKP
(Appendix B.11)
B.6 Population size (function NEWPOP) and reproduction pool selection (subroutine TOURN)

Having generated a full set of fitnesses for a parent population (Appendix B.4), the GA proceeds to selection and reproduction. Selection occurs in two steps:

- function NEWPOP (nowpop, minpop, maxpop, pstats, past):

  The first step is to determine whether the offspring population size is to change, from the current size NOWPOP, between size limits MINPOP and MAXPOP (users - declared in subroutine GenAlg), using the current and immediate past generation statistics in the arrays PSTATS and PAST, respectively. This is described in Section 4.3.

- subroutine TOURN (ncomp, nwin, ngene, npar, nchild, parfit, parent, CHILD):

  Having got a new offspring population size NCHILD, then tournament selection will put nchild chromosomes (each of length NGENE genes), into the columns of CHILD (ngene, nchild). These will be selected from the NPAR parent chromosomes in PARENT (ngene, npar), each of those having the corresponding fitness in PARFIT (npar) (no fitnesses are allocated to the offspring pool as they may be modified by reproduction). Selection will be by tournament with groups of NCOMP competitors leading to NWIN victors for each group (Section 3.2)
Set up LIST of randomly ordered pointers to all population members

Initialise offspring counter for tournament
noff:=0

While offspring still required do
noff<nchild

For current round, initialise number yet to compete at parent population size
nleft:=npar

Set competitor group size and number of victors - usually as specified but may not fit exactly near end of population

close to end - values may need adjustment

If within 2 groups of end
nleft<2*ncomp

Otherwise make 2 roughly similar groups

Still able to give new victors?

Usual nfight := ncomp

Take all remainder if not enough to split: nfight:=nleft

endif

endif

endif

A
B
C
D
E

See Section 2.4
- Subroutine
- NKLIST (App B.10)
Usually 
\text{nfight}:=\text{ncomp} 
\text{nvic}:=\text{nwin} 

And adjust no of winners to suit any changed 
\text{nfight}<>\text{ncomp} 
(if \text{nvic}<1 then \text{nvic}:=1) 

endif 

Randomly select group of \text{nfight} contestants from parents not yet competed in this round 

If more left than contest size 
\text{nleft}<>\text{nfight}?

Randomly select \text{nfight} from those not yet competed in \text{LIST}(1 \text{ to } \text{nleft}) 
\text{nleft}:=\text{nleft}-\text{nfight} and leave those to compete in this round in \text{LIST}((\text{nleft}+1) 
\text{ to } (\text{nleft}+\text{nfight}))

Only remainder left to fight so self-identifying stop condition 
\text{nleft}:=\text{nleft}-\text{nfight} 

Subroutine PKPART (App B.10) 

\text{nfight} competitors in this contest now in \text{LIST}((\text{nleft}+1) \text{ to } 
(\text{nleft}+\text{nfight})) so use their fitnesses in parfit to pick \text{nvic} offspring 

For req'd number of winners 
For \text{j}=1, \text{nvic}, 1 \text{ do} 

Store jth value as trial for best 
(initially last) before sorting by 
comparison with rest
start with next
For i:=(top+1),
(nleft+nright), 1 do
and do rest
Compare fitnesses
If better then this becomes best instead
enddo
Move jth best to top of range in LIST by exchange
Create offspring and increment noff count
label 104
enddo
label 105
Until none left to do
or population complete:
nleft=0 or noff=nchild?
endwhile
return

74
B.7 Reproduction - cross-over (subroutine XOVER)

subroutine XOVER (pcross, ncross, npoint, ngene, npop, KCROSS, POP):

The form of cross-over implemented has been outlined above in Section 3.3. A population pool of NPOP chromosomes (each with NGENE genes) in the columns of POP (ngene, npop) may be modified by overwriting, the number modified being recorded in KCROSS. The probability that modification (i.e. cross-over) occurs is PCROSS for each exclusive randomly selected group of NCROSS chromosomes (2 < ncross < npop). If cross-over does occur, it will be at NPOINT randomly selected points along the chromosomes:
Initialise cross-over counter for this generation so far:
\( k_{\text{cross}} = 0 \)

Create a randomly-ordered pointer list of all members who have not yet participated (all at this stage)

Initially the number left to cross-over is the whole population
\( n_{\text{left}} = n_{\text{pop}} \)

While still a pair left to go, proceed through remaining population
\( n_{\text{left}} > 1 \)

Cross-over in groups of \( n_{\text{cross}} \) unless last group is less

Get \( i_{\text{cross}} \) of those who have not yet participated in list (\( n_{\text{left}} + 1 \) to 
\( n_{\text{left}} + i_{\text{cross}} \)) and decrement:
\( n_{\text{left}} = n_{\text{left}} - i_{\text{cross}} \)
so that those pointers can be used to identify candidates for cross-over

If cross-over occurs, do it Random <= \( p_{\text{cross}} \)

Get \( n_{\text{point}} \) ordered cross-over locations in range 1 to \( (\text{ngene} - 1) \) in \( j_{\text{cross}} \)

Subroutine \( \text{PKPART} \)
(App B.10)

Function \( \text{RANDOM} \)
(App B.11)

See Section 2.4
-subroutine \( \text{MARKMARK} \)
(Appendix B.10)
and to ensure that cross-over terminates at end of chromosome set:
\[ j\text{cross}(\text{npoint}+1) := \text{ngene} \]

Count through \( j\text{cross} \) in jumps of 2 (start + end) to get sections of chromosomes that are crossed (intermediate sections unchanged)
For \( i := 1, \text{npoint}, +2, \text{do} \)

For sections that are crossed, do this for all chromosomes in group by rotation
For \( j := (j\text{cross}(i)+1), (j\text{cross}(i+1)), \text{do} \)

Temporarily store genes from first chromosome
\[ \text{store}(j) := \text{pop}(j, \text{list}(\text{nleft}+1)) \]

Then shift genes up from next \((i\text{cross}-1)\) chromosomes

For \( k := 2, i\text{cross}, +1 \) do

\[ \text{pop}(j, \text{list}(\text{nleft}+k-1)) := \text{pop}(j, \text{list}(\text{nleft}+k)) \]

Finally put stored genes into last chromosome
\[ \text{pop}(j, \text{list}(\text{nleft}+i\text{cross})) := \text{store}(j) \]
Increment count of those crossed
$kcross := kcross + 1$

enddo

label 1004

endif

endwhile

return
B.8 Reproduction - mutation and inversion (subroutines MUTATE, MUTCHR, INVERT and INVCHR)

For these chromosome-operators (Section 3.4) two top-level routines scan the population to decide whether the operation is going to occur for any individual chromosome:

- subroutine MUTATE (pmut, ngene, npop, K MUT, POP)

- subroutine INVERT (pinvrt, ngene, npop, KINVRT, POP)

For a population of NPOP chromosomes (each with NGENE genes) in the columns of POP (ngene, npop), each of these looks at each member with a chromosome-probability PMUT or PINVRT to decide whether mutation or inversion respectively, will occur and then counts the number of the operations at the generation in K MUT and KINVRT (additional operations may also occur because of subroutine UNIQUE - see Appendix B.3).

The above call, respectively:

- subroutine MUTCHR (ngene, CHROM)

- subroutine INVCHR (ngene, CHROMO)

to carry out the required chromosome-operation on the NGENE gene chromosome CHROM (ngene) or CHROMO (ngene). Note that subroutine NEWCHR (see Appendix B.2 above) has a similar parameter structure so that these three subroutines could be interchanged in certain places, eg. subroutines UNIQUE (Appendix B.3) or RNSIMR (Appendix B.4) (see Sections 4.3 and 4.4).

Subroutine MUTCHR simply choses a gene at random and calls user-defined subroutine MUTGEN to operate on this (see Appendix B.2 above), so that various mutation operations can be programmed (see Section 3.2 above). Inversion by subroutine INVCHR is fully defined, however (see Section 3.4 above):
Get different start then end gene locations for inversion

Randomly pick start location
istart in range 1 to n_gene

Function
RANDOM
(Appendix B.11)

label 2301
repeat

Randomly pick end location
i_end in range 1 to n_gene

Function
RANDOM

If same replace

F
Until i_end
different
to i_start
T

Get total number of genes in sequence for inversion

i_total := (i_end - i_start + 1)

i_total < 1?
T
i_total := i_total + n_gene
F

NB: count of number of gene swaps = trunc(i_total/2)

Identify exchange start points (with possible wrap-around at gene ends)
starting at either side (iexl and iexr) of partition point

To right of leftmost gene to be swapped
iexl := i_start + trunc(i_total/2)

A

80
Then to left of rightmost gene
iexr = istart + trunc((itotal-1)/2)

For the req'd number of gene swaps
For i = 1, trunc(itotal/2), 1 do

Get locations of each pair to be swapped (with wrap-around)

ileft = iexl - i

ileft < i?

F

T

ileft = ileft + ngene

iright = iexr + i

iright > ngene?

F

T

iright = iright - ngene

Do gene pair swap between ileft and iright

Label 2002

Enddo

Return
B.9 Recording and outputting results (subroutines POPOUT and RESULT)

GA output occurs in only four subroutines where it may easily be removed or modified (Section 2.5):

- subroutine GenAlg (see Appendix B.1 above);

- subroutine INIPOP (see Appendix B.3 above);

- subroutine POPOUT (igen, ngene, npop, lpopra, pop, popfit, pstats); and

- subroutine RESULT (igen, maxgen, ngene, ncream, ccream, fcream, lcream, ievol, revol, cevol).

Subroutine POPOUT outputs population information at every generation, most of which is lost at the next generation, whereas subroutine RESULT outputs after the GA has terminated, information that has been stored by:

- subroutine GenAlg (see Appendix B.1 above); and

- subroutine cream (see Appendix B.5 above).

Subroutine POPOUT at generation number IGEN, points out the current generation population of NPOP chromosomes (each with NGENE genes) stored in the columns of POP (ngene, npop) with corresponding member fitnesses in POPFIT (npop) whose rank pointers are in LPOPR A (npop) and (see Section 2.3) for current population fitness statistics in PSTATS (6). To avoid it, simply comment-out the call in subroutine GenAlg.

Subroutine RESULT, after the GA has terminated at generation IGEN out of a maximum of MAXGEN, prints out a record of the GA operation at each generation using counts stored in IEVOL(7, maxgen) population fitness statistics stored in REVOL (5, maxgen) and the best chromosome (each with NGENE genes) at each generation stored in CEVOL (ngene, maxgen). It then prints out the NCREAM best chromosomes found and stored by subroutine CREAM in CCREAM (ngene, ncream) with their corresponding fitnesses FCREAM (ncream) using the rank pointer LCREAM (ncream) to order them (see Section 2.3).

Subroutine GENALG accumulates the data stored in IEVOL, REVOL and CEVOL which are output by subroutine RESULT. Subroutine GenAlg effectively does little. Integer counter parameters to the GA operation subroutines it calls automatically form all but one row of IEVOL (though for the first generation it initialises three of these to ensure tidy output). A real array parameter to subroutine postat also automatically forms REVOL.
B.10 Pointer integer vectors - subroutines PKPART, MKLIST, MKMARK and MKSEQL

The GA operations make use of integer pointer arrays (see Sections 2.3 and 2.4) and a series of subroutines generate and manipulate these.

(a) Subroutine MKSEQL (min, incr, nlist, LIST):

This is the simplest and makes a sequential (next highest follows) list of NLIST integers in LIST (nlist), with a lowest value MIN and subsequent values at increments INCR up to a maximum = (min + (nlist - 1) * incr), with a randomly chosen first value in list (1) and wrap-around, eg, nlist = 10, min = 3, incr = 2 (and hence maximum = 21 might give, if 7 was the randomly chosen first element):

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10 = nlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>list (i)</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>21</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

In practice it is only used with incr = 1 at present.

(b) subroutine MKLIST (nlist, LIST):

This fills an integer vector LIST (nlist) with NLIST exclusive but randomly-ordered pointers in the range 1 to nlist (increments of 1), eg, for nlist = 7:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>list (i)</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

in random order.

(c) subroutine MKMARK (min, max, nmark, MARKER):

From the integers in the range MIN to MAX (increments of 1), randomly chooses NMARK of these in the first nmark elements of MARKER (max) and orders then (low first/high last), with the remainder left randomly ordered. Eg., for min = 1, max = 10, nmark = 4 initially start with (max-min + 1) = 10 elements which might be

6 7 8 9 10 12 3 4
then randomly select any nmark = 4 of these and swap them into the first fmark = 4 elements:

$$8\ 3\ 1\ 10\ \big|\ 9\ 6\ 2\ 7\ 4$$

and finally sort the first nwork = 4 into ascending order but leave the rest unchanged:

$$1\ 3\ 8\ 10\ \big|\ 9\ 6\ 2\ 7\ 4$$

(d) subroutine PKPART (npick, nlist, NLEFT, LIST):

LIST (nlist) is an integer vector of NLIST exclusive pointers in the range 1 to nlist (increments of 1), but positioned so that the first NLEFT elements (index 1 to nleft) have some significance (i.e. have not previously been chosen) whereas elements (nleft +1) to nlist have previously been randomly chosen. PKPART

A. Randomly chooses npick from the nleft not previously chosen.
B. Moves them by exchange to locations (nleft -npick +1) to nleft.
C. Then resets nleft = (nleft - npick)
D. So that the elements chosen this time are at (nleft +1) to (nleft + npick) and elements 1 to nleft have not yet been chosen (ready for next call).

Eg, nlist = 11 and say initially nleft = 7 with npick =3:

Say initially:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>list (i)</td>
<td>9</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>11</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

nleft = 7

then finally

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>list (i)</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>11</td>
<td>9</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

nleft = 4  npick = 3  previously chosen
Unlike the others (A - C) above, this does not self-initialise the pointers, but assumes that they already exist and modifies them.
For each of npick elements this time:
For i:=1, npick, +1 do

Get random pointer in range of
as yet unchosen 1 to (nleft+(i-1))
jran:=1+round(RANDOM*(nleft-i))

Then swap chosen element with element
(nleft-(i-1)) so that latter remains
available for selection next time

Finally decrement partition pointer
when selection is completed
nleft:=nleft-mpick

nleft is the partition
pointer - it records
the division between
selected and unselected
B.11 Standard numerical routines - functions CHSAME and RANDOM, subroutines ISORTI and IRANKP

The last three of these could probably be replaced by standard published procedures, eg. Press et al. (1992)

(a) function CHSAME (ngen, chrom1, chrom2):

Compares two chromosomes (both of length NGENE genes) CHROM1 (ngen) and CHROM2 (ngen) gene-by-gene to see if they are identical, returning true if they are and false if not. Does this by subtraction so that a tolerance GENTOL can be applied in practical applications (currently GENTOL = 0.0 for exact identity).

(b) function RANDOM

A floating-point form of the Knuth subroutine method with the features:

- self-initialising (i.e. no separate “warm-up” routine, uses its own counter IRCALL to identify the first call);

- no parameters, some called from many subroutines at different levels in the code;

- hence “seed” values (SEED, START) declared locally; and

- the counter (IRCALL) and necessary Knuth array KNUTH (55) are stored globally at the level of subroutine GENALG (for one complete GA cycle) using COMMON block /RDM/.

This could be replaced by published codes (eg. Press et al. 1992).

(c) subroutine ISORTI (nsort, ntotal, LIST):

Given an integer vector LIST (ntotal) with NTOTAL elements, sorts the first NSORT of these (i.e. elements 1 to nsort) into ascending order using the standard straight insertion method. This is a standard and well-documented method with the only non-standard feature being the ability to restrict the sort range using the “extra” parameter nsort. For nsort relatively large (say nsort > 20), then other sorting methods are probably faster, but their coding is more complex (this could be updated in future).

(d) subroutine IRANKP (nval, rval, POINT):

Given a vector RVAL (nval) with NVAL real elements, this produces an integer rank pointer index POINT (nval) which gives the indices of rval for values in ascending order without re-ordering rval. This uses Quicksort (without
recursion) until fewer than NSORT (currently nsort = 7) are left when straight insertion takes over, following a standard and well-documented procedure. This could be replaced by published code, e.g. subroutine INDEXX from Press et al (1993).
APPENDIX C  GOLDBERG $x^2$ EXAMPLE OUTPUT

This contains:

(i) The main program used to call subroutine GENALG for this case, and

(ii) the output generated by the code as presented in Appendix A. Discussion
     of these above aspects is given in Section 5.
Appendix C

Trivial test program to run Goldberg example:

```
PROGRAM GATest
C Test GA implementation
INTEGER input
REAL output(4)
CALL GenAlg_SETUP()
PRINT 'Test of GenAlg completed'
STOP

END
```

Typical output from code above with Appendix A:

```
SET UP WITH FOLLOWING GA OPTIONS:
Reproduction: Tournament in groups of 4 competitors with 1 victor(s)
Cross-over at probability 1.00000 in groups of 2 chromosomes
Mutation at probability 1.00000
Inversion at probability 1.00000
Maximum number of generations = 5
Population: Chromosome length = 5 genes
Variable population from 7 to 8 starting at 8 members

INITIAL RANDOMLY GENERATED SEED POPULATION OF 8 NUMBERS IS:
Seed chromosome 1 is:
0.00000 1.00000 0.  0.  0.00000
Seed chromosome 2 is:
0.  0.0  0.0  0.  0.0
Seed chromosome 3 is:
1.00000 0.  0.00000 0.  0.
Seed chromosome 4 is:
0.  0.00000 1.00000 0.  0.00000

Population for generation = 1
Average Median Standard Deviation Absol
Fitness Fitness Deviation Deviation
398.5000 394.5000 273.0500 214.0000
Rank = 1 with fitness = 425.000
1.00000 1.00000 0.  0.  0.00000
Rank = 2 with fitness = 402.000
1.00000 0.  0.00000 0.  0.
Rank = 3 with fitness = 169.000
0.  0.  0.00000 1.00000 0.00000
Rank = 4 with fitness = 0.
0.  0.  0.  0.  0.

Population for generation = 2
Average Median Standard Deviation Absol
Fitness Fitness Deviation Deviation
449.0500 448.0000 141.9300 153.0000
Rank = 1 with fitness = 425.000
1.00000 1.00000 0.  0.  0.00000
Rank = 2 with fitness = 402.000
1.00000 0.  0.00000 0.  0.
Rank = 3 with fitness = 304.000
1.00000 0.00000 1.00000 0.00000
Rank = 4 with fitness = 0.
0.  0.  0.  0.  0.
```

Appendix C

```

1.00000 0.  1.00000 1.00000 0.
Rank = 1 with fitness = 400.000
1.00000 0.  1.00000 0.  0.
Rank = 2 with fitness = 244.000
1.00000 0.  0.  0.  1.00000

Population for generation = 3
Average Median Standard Deviation Absol
Fitness Fitness Deviation Deviation
496.2200 496.0000 268.0269 194.6630
Rank = 1 with fitness = 490.000
1.00000 1.00000 1.00000 1.00000 0.
Rank = 2 with fitness = 626.000
1.00000 1.00000 0.  0.  1.00000
Rank = 3 with fitness = 400.000
1.00000 0.  0.00000 0.  0.
Rank = 4 with fitness = 244.000
1.00000 0.  0.  0.  1.00000
Rank = 5 with fitness = 296.000
1.00000 0.  0.  0.  0.  0.

Population for generation = 4
Average Median Standard Deviation Absol
Fitness Fitness Deviation Deviation
787.9900 712.1000 118.7238 83.0030
Rank = 1 with fitness = 900.000
1.00000 1.00000 1.00000 1.00000 0.
Rank = 2 with fitness = 841.000
1.00000 1.00000 1.00000 0.  1.00000
Rank = 3 with fitness = 784.000
1.00000 1.00000 1.00000 0.  0.
Rank = 4 with fitness = 625.000
1.00000 1.00000 0.  0.  1.00000

Population for generation = 5
Average Median Standard Deviation Absol
Fitness Fitness Deviation Deviation
715.0000 746.0000 218.2851 175.1090
Rank = 1 with fitness = 621.000
1.00000 1.00000 1.00000 1.00000 1.00000
Rank = 2 with fitness = 552.000
1.00000 1.00000 1.00000 0.  1.00000
Rank = 3 with fitness = 794.000
1.00000 1.00000 1.00000 0.  0.
Rank = 4 with fitness = 784.000
1.00000 1.00000 1.00000 0.  0.
Rank = 5 with fitness = 400.000
1.00000 0.  1.00000 0.  0.
```
### Appendix C

**Evolving Process Over 3 to 5 Generations**

<table>
<thead>
<tr>
<th>Generation</th>
<th>Population Size</th>
<th>No. of Seed Diploid Chromosomes</th>
<th>No. of Seed Tetraploid Chromosomes</th>
<th>No. of Diploid Mutations</th>
<th>No. of Tetraploid Mutations</th>
<th>No. of Fitter Mutations</th>
<th>No. of Fitterer Mutations</th>
<th>Best Chromosome of Generation</th>
<th>Fitness</th>
<th>Standard Deviation of Fitness</th>
<th>Best Chromosome of Generation</th>
<th>Fitness</th>
<th>Standard Deviation of Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80</td>
<td>500</td>
<td>500</td>
<td>120</td>
<td>120</td>
<td>100</td>
<td>100</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>500</td>
<td>500</td>
<td>120</td>
<td>120</td>
<td>100</td>
<td>100</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>80</td>
<td>500</td>
<td>500</td>
<td>120</td>
<td>120</td>
<td>100</td>
<td>100</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>500</td>
<td>500</td>
<td>120</td>
<td>120</td>
<td>100</td>
<td>100</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>80</td>
<td>500</td>
<td>500</td>
<td>120</td>
<td>120</td>
<td>100</td>
<td>100</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0</td>
</tr>
</tbody>
</table>

**Best 4 Chromosomes From All Generations**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Chromosome 1</th>
<th>Fitness</th>
<th>1.00000</th>
<th>1.00000</th>
<th>1.00000</th>
<th>1.00000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>