Index Tracking : Genetic Algorithms for Investment Portfolio Selection

J Shapcott

EPCC–SS92–24

Index Tracking : Genetic Algorithms for Investment Portfolio Selection

J Shapcott

EPCC–SS92–24

September 1992

Abstract

This project was concerned with passive portfolio selection using genetic algorithms and quadratic programming techniques. Searching a large universal set of shares for a subset that performs well is intractable, so a stochastic search method must be used. The genetic algorithm generates the subsets, and quadratic programming is used to find both their performance and the proportion of the available capital that should be invested in each member company. Separate subpopulations are maintained on different processors of a Meiko Computing Surface, with occasional migration of genomes. This strategy allows several differing threads of the search to be pursued within the separate subpopulations, and migration encourages convergence to the global optimum, instead of local optima.

1 Introduction

Index Tracking is a method of passive portfolio management. It attempts to match the performance of a theoretical portfolio, such as the FTSE-100 index, as closely as possible. This approach is used by fund managers when they do not feel confident enough of out-performing the market, and are content to follow the average performance.

Matching the performance of an index can be performed two ways. The first is *full replication*, in which an investment is made in every every constituent of the index, proportional to its market share. This achieves a perfect match, but incurs high initial transactions costs, and is difficult to rebalance when changes are made to the index, for example, the issue of a new set of shares. The other approach is *partial replication*, in which an investment is made in a small proportion of the shares, while attempting to match the performance of the entire index. This incurs lower initial transaction costs, and is easier to rebalance. It also allows the investor to constrain the choice of investment that is made, by insisting on the inclusion or exclusion of some companies, or by setting the proportion of the capital that is to be invested in others.

Partial replication, however, introduces a Tracking Error, the measure of the deviation of the chosen portfolio from the index. The aim of fund managers is minimise this tracking error. This can be expressed as a quadratic programming problem. The overall approach is that given a subset of constituent shares of the index, the tracking error is the expected squared deviation of return from that of the index. The proportion of capital to be invested in each company is calculated as part of the same problem.

This requires two pieces of data, the covariance matrix defining the relationship between all of the companies in the index, and the capitalisation weights. The covariance matrix shows the correlations that exist between companies in the index. A large positive value occurs when the return from two companies follow very similar trajectories. A small negative value occurs when two companies follow roughly opposing trajectories. The capitalisation weights are simply the normalised returns for the index. They are calculated by taking the return for each company, and dividing it by the sum of the returns for all companies.

The problem, then, is to select a subset of all of the shares in an index which best matches the performance of the market as whole. The selection of subsets is algorithmically intractable, so stochastic search systems, such as *genetic algorithms* suggest themselves.

The genetic algorithm will be used to select subsets of shares, and quadratic programming will perform the index tracking, evaluating the performance of each particular subset.

The following report will discuss how index tracking can be expressed in quadratic programming terms, and genetic algorithms in general. This will be followed by a description of how certain operations define the power of genetic algorithms. The final sections discuss the testing, the results achieved, and what conclusions can be drawn from them.

2 Index Tracking

Quadratic programming is a minimisation procedure for quadratic functions, those containing only quadratic terms, linear terms, and a constant. For a given a set of variables, the function can be encoded as a matrix containing the coefficients of the quadratic terms, a vector containing the coefficients of all the linear terms, and the constant. The variables defined in the equations below refer, then, to these vectors and matrices.

The function which needs to be minimised in the index tracking problem is given below. The symbols are defined as follows,

X The vector containing the proportion of capital to be invested in each member of the index.

H The vector containing the capitalisation weight of each member of the index.

G The two dimensional matrix defining the correlations between members of the index. This matrix, naturally, is symmetric.

The objective function is given by,

$$f(X) \quad = \quad (X - H)^T G(X - H)$$

This can be expanded to,

$$f(X) \quad = \quad X^T GX - 2H^T GX + H^T GH$$

The first term of the equation gives rise to the quadratic terms of the equation, the second to the linear terms, and the third to a constant. The constant is not strictly necessary for function minimisation.

The linear coefficients and the constant are precomputed using these matrices for the entire index. However, the problem can be simplified when minimising the function for subsets, by only including those coefficients relevant to the subset being evaluated. The quadratic programming code can be given only the coefficients which refer directly to members of the subset, and will return the minimum defined by the equation above, as well as the vector X. This makes for considerable savings in terms of both computation time, and memory usage.

Index tracking was implemented using Fortran77 code supplied by R. Fletcher of the Mathematics Department of the University of Dundee. The process of index tracking, and the necessary quadratic programming is discussed at length in [Kwiatkowski, 1991].

3 Genetic Algorithms

Genetic algorithms are stochastic optimisation techniques invented by John Holland [Holland, 1975]. They use ideas taken from biology to guide the search to an optimal, or near optimal, solution. The general approach is to maintain an artificial ecosystem, consisting of a population of chromosomes. Each chromosomes represents a possible solution to the general problem. In this case, they take the form of a representation of a particular subset of shares from the index. Attached to each chromosome is a fitness value, which defines how good a solution that chromosome represents. By using mutation, crossover (breeding), and natural selection, the population will converge to one containing only chromosomes with a good fitness. The fitness function in this case is the subset's tracking error, and the object of the search is to find chromosomes with high fitness values.

Many issues arise during the construction of a particular genetic algorithm, all of which affect its performance. Crossover operators, which take two parent chromosomes and combine them in such a way as to produce a child, need to be carefully designed, to allow the transmission of the best properties of the parents to the child. Mutation is necessary to prevent areas of the search space being discarded, but too high a mutation rate will prevent the desired convergence. The method that is used to update the population in each cycle of the algorithm is another factor. *Generational update* uses portions of the population to generate enough children to replace the population in its entirety, but more fine grained approaches replace just a few members of the population with children.

The most important of these factors is the design of the crossover operator. This defines the exploratory power of the search, since it is the function that gives rise to the vast majority of new chromosomes. Using mutation alone is not usually sufficient, because of the difficulty of setting a mutation rate which allows for a rapid search, while preventing the search from becoming completely random. Crossover also allows two good chromosomes representing good partial solutions to be combined to form a child representing an even better, more complete, solution.

For further information, refer to [Goldberg, 1989].

4 Random Assorting Recombination

The particular crossover operator used by this project is a form of the Random Assorting Recombination (RAR) operator, defined in [Radcliffe, 1992]. This has several beneficial properties, allowing for a flexible and reliable transmission of genes from parents to children.

The reasoning behind the RAR operator for subset selection, is based on the the ideas given in [Radcliffe, 1991] for ideal crossover operators, which are independent of both the specific problem, and the representation used.

The following paragraph makes use of two terms borrowed from biology, *gene*, and *allele*. A gene is a piece of genetic information that can defines some property of an individual, for example, eye colour. An allele is the value that a gene could take, for example, in the case of the eye colour gene, it could take the value “blue”, or “brown”.

Each member of the universal set, in this case, all of the members of the index, represent a gene in the chromosome. Each gene may have one of two alleles, membership, or non-membership. The ideal crossover operator should possess two properties, *respect*, and *assortment*. Respect embodies the idea that if the parents share alleles, then those alleles should be present in any children. For example, if both parents include in their set the item referring to British Petroleum, and neither parent includes the item referring to Glaxo, the the child should include British Petroleum, and exclude Glaxo. Assortment embodies the idea that any alleles possessed by the parents should be available to the child, as long they are consistent with each other. For example, if one parent includes British Petroleum but excludes Glaxo, and the other parent includes Glaxo but excludes British Petroleum, then the child should be able to make up any combination of membership or non-membership of either of those companies, assuming that these combinations do not lead to illegal chromosomes.

For variable size subsets, it is possible to design an operator with both of these properties. However, the problem here uses subsets of a fixed size. If variable sized subsets were allowed, then the optimum would occur when the subset of shares was the same as the universal set, i.e. full replication, giving rise to no tracking error at all. Constraining the size means that respect and assortment become incompatible. An example will demonstrate this. Suppose there are two parents, of a fixed size five, that share four members. Respect demands that those four members should be in the child, but assortment states that any combination of the members of either parent should be available to the child. After placing the four shared members into the child’s subset, we have room for only one more member. This means that we have to include the remaining member from one parent, but exclude the one from the other parent. It can be seen that only one or the other of the non-shared members is available to the child, violating the assortment condition.

It is possible to approximate both of these conditions to some extent, even for fixed size subsets, using the RAR operator.

The RAR operator not only takes two parents as arguments, but also a weight. This weight is an integer value, and denotes the amount of importance respect is given over assortment. If there is a high weight, then respect is given priority over assortment. The idea behind the operator is quite simple.

Given two parents, all of the alleles are placed in a bag. If a parent shares an allele, the membership or non-membership of the same member of the universal set, then a number of copies of the allele, equal to the weight, are dropped into the bag. If an allele is not shared, one parent including a member the other excludes, then one copy of the non-membership allele is dropped in the bag, followed by one copy of the membership allele. This is performed for all genes, so that there is an allele defined for each member of the universal set.

Once the bag has been filled, alleles are drawn out at random and assigned to the child. If a membership allele is drawn, then that member is placed in the child, if a non-membership allele is drawn, then that member is barred from the child. If an allele is drawn for a gene that has already been assigned an allele, then it is ignored. This process terminates when the child contains the maximum number of members, or when the maximum number have been barred, leaving the child with the unused members of the universal set.

It can be seen that if the weight is high, then shared alleles are more likely to be drawn from the bag, which shows respect. However, it is still possible to draw non-shared alleles from the bag, and place them in the child, which shows assortment. The “natural” weight is 2.

5 RPL/Framework

The project was implemented using RPL/Framework, which is a general purpose tool for implementing genetic algorithms. This has been developed within the Edinburgh Parallel Computing Centre, [Russo, 1991]. It acts as a parser for programs written in *Reproductive Plan Language* (RPL), an interactive environment suitable for experimentation, and provides a means for embedding problem specific representations and operators into RPL.

One major problem encountered when using genetic algorithms is premature convergence. This occurs when the population becomes homogeneous, without having found the optimal solution. The search becomes dependent on the mutation operator at this point, and since low mutation rates are used, it is most unlikely that any considerable improvement will be made.

Framework uses isolated subpopulations on separate processors. The subpopulations act as ordinary genetic algorithms, but will tend to speciate, by following different promising avenues of the search. Each one may converge prematurely, but this can be countered by allowing the occasional migration of genomes between subpopulations. The combination of good solutions from different species may lead to even better solutions, but even if this is not the case, the heterogeneity of the overall population is maintained.

The particular genetic algorithms used by this project have been implemented using RPL, and consist of the operators required by the genome representation used for sets. Each operator is specified by the programmer to RPL as two C language functions, which are built into the Framework at compile time. The first C language function is called during the execution of the RPL program, and performs the actual operation on the genomes.

The second C language function organises the arguments that an operator may need, for example, the RAR operator requires an argument for the weight it is to use. It is called during the first phase of execution within Framework, which compiles the RPL program.

Framework provides a number of representation independent operators, which allow for the maintenance of the population. Examples include operators which select parents, according to some scheme which takes account of the fitness of the population members, and place them on a stack. Other operators remove members from the stack and place them, according to another scheme, into the population. There are also operators which allow for the scaling of fitness values. This is important since when a population tends towards homogeneity, all the members have very similar fitness values. It becomes increasingly difficult to select a particularly fit member for parenthood. Scaling the fitness values allows this distinction between fitness values to be maintained. There are other operators which allow for statistics about the population to be gathered at regular intervals.

The last class of standard operators are those which deal with the migration of genomes between subpopulations on separate processors.

For this project, genomes are represented by an array of integers. Each integer represents a member of the universal set, and is used by the quadratic programming code to index into arrays representing the matrices required by the quadratic code, stored for the universal set.

The representation dependent operators required for the construction of a genetic algorithm are ones to generate a random genome, evaluate it using the quadratic programming code, mutate it, perform random assorting recombination, and save it to a file.

6 RPL Programs

Three separate RPL programs were used for this project. One is an implementation of random search, The other two are genuine genetic algorithms, one with isolated the subpopulations, and one which allowed the migration of genomes.

Random search is a search strategy in which random solutions are continually generated, saving the best one that has been found so far. This is a very robust strategy, and is useful as a control. Comparisons with random search are used to show that the more complicated genetic algorithms are actually benefiting from their sophistication.

The two genetic algorithms use a fine grained update. The following process is repeated until the number of children equal to the population size have been generated.

1. Select two parents using binary tournament. This selection scheme looks at two members of the population at random, and with a supplied probability selects the better of the two, otherwise, it selects the worse of the two.

2. Produce a child of these parents using the RAR operator.

3. With a low mutation rate, mutate this newly generated child.

4. Replace the worst member of the population with the child.

This process will be called a *generation* for the rest of this report. This is technically an abuse of the term, and should not be confused with *generational update*.

This entire process is repeated for a certain number of generations. Population statistics are gathered at regular intervals.

The only difference between the two genetic algorithms is that in the one that allows migration, immigrants are collected together and replace the worst members of the population before a generating any new children. After they have been generated, the best member of the population is located, and allowed to migrate to one of the other subpopulations.

This strategy allows the subpopulations to speciate, before introducing a different species of genome. This maintains the heterogeneity of the entire population, but still allows for convergence, which could well be disrupted if migration was permitted after the generation of every child.

7 Implementation

The programming that was required by the project falls into four main stages. Each stage was tested before moving onto the next stage, but several revisions of previous work had to be made.

The first of these was to design and implement a set representation appropriate for genetic operators. Each genome takes the form of a C language structure, an array of integers, and an integer giving the current size of its subset. The maximum allowed size of any subset is an integer variable available only to the file containing the genetic operators and the C language wrapper for the quadratic programming code. Simple set operations were implemented: a test for membership of a particular element being within the set, a function to add elements to the set, one to remove elements, one to see if the set is empty, and one to see if the set is of maximum size. Members of the set are stored in ascending order. This allows the membership function, which is the one called most frequently, to be binary search, giving an improved execution time over linear search.

All of the genetic operators, such as generating a random genome and mutation, were implemented using this group of set functions. The most complicated was the random assorting recombination operator.

The next stage was to build the quadratic programming code into an evaluation operator. This had been written in Fortran77, whereas the rest of the project was written entirely in C. Cross-calling between the two languages was completed, with some small problems due to differences between the languages. An example of this is that Fortran always passes variables by reference, but C always passes them by value. Another, is that arrays in C are always indexed starting from the major axis, but in Fortran, arrays are indexed starting from the minor axis. A number of bugs within the Fortran code were discovered during the testing, and fixed. The first few were simple spelling mistakes in variable names, picked out by the compiler's type checker. Another one was more serious. The original programmer used one of the variables as a real number in one subroutine, and as an array of characters in another. This was not shown up by the type checker, and caused address exception errors. Replacing the real number variable with an array of characters during the subroutine call seemed to fix the problem. The final problem was due to write statements. If a Fortran subroutine is called from a C function, any attempt at input or output while inside the fortran will cause a bus error, and abort execution. This was solved by commenting out all of the Fortran write statements.

Once this code was running, It took a great deal of effort to find parameters that would give results reliably. Simple functions were used for this testing. The minima, which it is designed to find, should always be greater than zero, but it was found that the absolute minimum, beyond which the quadratic programming would abort, had to be set to -1. The linear constraint, which states that the vector X, should always sum to 1, caused many problems. In a few cases, evaluation would fail completely, returning a negative minima. By fixing the lower bound on this linear constraint to 1, but the upper bound to 10^{30}, these problems disappeared. The minima always exists at the point were the sum of the vector X is 1, so the upper bound can legitimately be set to any number greater than 1. A final, minor, problem was to make sure that all arrays used by the quadratic programming were set to zero before it was called.

A major problem was encountered when attempting to use genuine data from the FTSE-100 share index. The minima appeared to exist at the point when all of the capital should be invested in a single company. This took a long time to track down, but had a simple solution. The quadratic programming code needs to be set up with an initial estimate for values in the vector X. These were being set zero, but need to set proportional to the capitalisation for that company, within that particular subset.

The final stage was to integrate the genetic operators into Framework. This involved writing the two C language functions described earlier, for each of the genetic operators already written. This went very smoothly, with the exception of having to modify some of the existing standard operators. One example in particular was that the migration of genomes between processors was implemented so that migration would always be successful, when the success of a migration should have been probabilistic. These were then used to write the RPL programs described above (listings appear in Appendix A).

8 Testing

A simple objective function was used for testing the effectiveness of the three search systems, random search and the two genetic algorithms. This function takes a universal set of size N and a subset of size n, such that $n \leq N$. It returns a fitness equal to the maximum allowed subset size, minus the number of members which appear in the first n members of the universal set. The optimum is therefore 0, when the subset contains exactly the first n members of the universal set. This, while appearing simple to solve, has a search space of size $\binom{n}{N}$, and the number of solutions diminishes radically as the fitness improves, making it more difficult for stochastic solution than it first appears.

Each RPL program uses 16 subpopulations, and generates exactly the same number of random genomes in each cycle, and runs for exactly the same number of cycles, so the only difference in the exploratory power is due to the nature of the search, and not an increased sample size.

Figures 1- 6 show the behaviour of the three systems. There is a single graph for random search, and two for each the genetic algorithms. The remaining graph shows the increased resistance to premature convergence when migration is introduced into the genetic algorithm.

The function was evaluated with $n = 20$ and $N = 100$, for all three RPL programs.

Random search shows a reasonable performance (Figure 1), finding a best solution with a fitness of 7. However, only one of the subpopulations finds this solution, with most achieving a solution with a fitness of 8.

The genetic algorithm with isolated subpopulations performs somewhat better. 11 of the subpopulations achieve a solution with a fitness of 4, (Figure 2), and all of the rest achieve solutions with a fitness of 5. The stochastic nature of the search is shown well here, with the subpopulations all finding better solutions at widely different times. One of the populations find its best solution after about 700 generations, but some of them have still not found it after 2000.

The median shows the behaviour of the whole of the subpopulations, (Figure 3), being a measure of the direction all of their members are taking. It can be seen that the subpopulations all move towards better solutions at roughly the same time, but the time it takes between the first moving, and the last moving, increases as the solutions become better. This is due to the decreasing number of possible solutions. By the end of 2000 generations, 9 subpopulations have a median of 5, and the remaining have a median of 6.

Similar behaviour between the best solutions discovered can be seen when migration is introduced. (Figure 4). After 2000 generations, the best solution is still 4, and 10 of the subpopulations have attained it. The rest of the subpopulations all have best fitness of 5.

The differences are shown up by looking at the median fitness for migrating subpopulations (Figure 5). The time taken for all of the subpopulations to make the transition to a better solution is decreased by a large amount. This shows how the subpopulations have worked together, using the migrating genomes to behave much more like a single population.

Introducing migration does improve the resistance to premature convergence. The final graph entitled “Population Diversity” (Figure 6) shows the average difference between the subpopulation best and median fitnesses. This is a measure of how much of the population has converged towards the best score. A small difference means that more of the population has moved towards the best fitness. It can be seen that for the vast majority of the time, the difference is always larger for the genetic algorithm with migrating subpopulations.

9 Results

Once the genetic algorithm had been tested, the covariance matrix and capitalisation weights for the FTSE-100 share index were obtained, and run with using index tracking as the fitness function. The results may be slightly atypical since capitalisation data was obtained for a week in which The Midland Bank was bought out.

Figure 7 shows a comparison between the genetic algorithm using isolated subpopulations, and the one using migration. The results plotted show the best solution found in any subpopulation, the average of the best solutions found in separate subpopulations, and the average of the median across the subpopulations. This shows that, like the simple objective function, they follow very similar paths, and attain roughly the same results. The genetic algorithm that uses migration has been found to find a better solution in most cases.

Figure 8 shows that the genetic algorithm that uses migration resists premature convergence, but this resistance is less marked than with the simple objective function.

10 Conclusions

The only major weakness in the project was the failure to find the global optima for the simple objective function. It is not clear whether this is a weakness in the operator, or inappropriate parameter settings for the problem.

The remaining aspects of the project were fairly successful. The evidence gained from experiments using the simple objective function, showed that the use of random assorting recombination crossover, and RPL/Framework led to a flexible genetic algorithm, which performed significantly better than random search. It has also been shown that separate subpopulations, with occasional migration, do act as something of a safeguard against premature convergence.

A RPL Program Listings

A.1 Random Search

```
plan(Set)

   int processors;          %% number of slave processes
   int generations;         %% number of pseudo-generations to be produced
   int period;              %% frequency with which to gather statistics
   int minimize;            %% minimize the objective function
   string input;            %% input file containing FTSE-100 info
   string output;           %% output file for statistics
   int counter;             %% counter used by doi/untili pairs

   randomize;               %% set the seed for the random number generators
   read-data &input;        %% read the FTSE-100 data from the input file

   %% generate an completely random starting population
   doi &counter 1;
      random-genome;        %% generate a random genome
      bqpd 1;               %% evaluate it
      assign-pop &counter;  %% store it in the population
   untili &counter &popsize 1;

   cycle

      %% find the best solution in the population, and store it in the
      %% first position
      select-best &minimize;
      assign-pop 1;

      %% generate random genomes for the rest of the population
      doi &counter 2;
         random-genome;
         bqpd 1;
         assign-pop &counter;
      untili &counter &popsize 1;

      %% find the best one to send to the master process for analysis
      select-best &minimize;

      analysis
         %% evaluate all genome sent by slaves
         bqpd &processors;

         %% store them in the population
         doi &counter 1;
             assign-pop &counter;
         untili &counter &processors 1;

         %% every &period generations, save the statistics to the output file
         get-stats &output &period &minimize;
      endanalysis

   endcycle

   %% select the best solution for the final analysis phase
   select-best &minimize;

   analysis

      %% evaluate and store the genomes sent by the slave processes
      bqpd &processors;
      doi &counter 1;
          assign-pop &counter;
      untili &counter &processors 1;

      %% find the best one found by any of the slaves, and save it to
      %% the output file
      select-best &minimize;
      save-genome &output;

  endanalysis

endplan

set &processors 16
set &counter 1
set &minimize -1
set &generations 1000
set &period 50

set &input 'FTSE-100'
set &output 'random.out'

set &popsize 51
set &cachesize 0
set &stacksize &processors

go &generations
```

A.2 Genetic Algorithm for Isolated Subpopulations

```
plan(Set)

   int processors;            %% number of slave processes
   int generations;           %% number of pseudo-generations to be produced
   int period;                %% frequency with which to gather statistics
   int counter;               %% counter used by doi/untili pairs
   int minimize;              %% minimize the objective function
   int order;                 %% weight argument for the RAR operator
   double mutation;           %% mutation rate
   double aggro;              %% aggressiveness for binary tournament select
   string input;              %% input file containing FTSE-100 info
   string output;             %% output file for statistics

   randomize;                 %% randomize the random number generators
   read-data &input;          %% read the FTSE-100 info from the input file

      %% generate a completely random starting population
   doi &counter 1;
      random-genome;
      bqpd 1;
      assign-pop &counter;
   untili &counter &popsize 1;

   cycle

         %% generate &popsize children using the following method
      doi &counter 1;

            %% select two parents
         binary-tournament &aggro &minimize;
         binary-tournament &aggro &minimize;

            %% perform crossover, mutation and evaluation operations
         rar &order;
         mutate &mutation;
         bqpd 1;

            %% replace the worst member of the population with the new child
         replace-real-worst 1 &minimize;

      untili &counter &popsize 1;

         %% rank the population to gain a median value
      rank-population &minimize;

         %% select the best member of the population for analysis
      select-best &minimize;

      analysis

            %% evaluate the genomes sent by the slave processes, and store
            %% them in the population
         bqpd &processors;
         doi &counter 1;
            assign-pop &counter;
         untili &counter &processors 1;

            %% rank the population to get a median value
         rank-population &minimize;
```

```
         %% every &period generations, save the statistics to the output file
         get-stats &output &period &minimize;

      endanalysis

   endcycle

   %% select the best solution for the final analysis phase
   select-best &minimize;

   analysis

      %% evaluate and store the genomes sent by the slave processes
      bqpd &processors;
      doi &counter 1;
         assign-pop &counter;
      untili &counter &processors 1;

      %% select the best one found by any slave
      select-best &minimize;

      %% save it to the output file
      save-genome &output;

   endanalysis

endplan

set &counter 0
set &processors 16
set &minimize -1
set &order 2
set &mutation 0.001
set &aggro 0.6
set &input 'FTSE-100'
set &output 'isolate.out'
set &popsize 50
set &cachesize 0
set &stacksize &processors

set &generations 1000
set &period 50

go &generations
```

A.3 Genetic Algorithm for Migrating Subpopulations

```
plan(Set)

   int processors;          %% number of slave processes
   int generations;         %% number of pseudo-generations to be produced
   int period;              %% frequency with which to gather statistics
int counter;              %% counter used by doi/untili pairs
int minimize;             %% minimize the objective function
int order;                %% weight argument for the RAR operator
double mutation;          %% mutation rate
double migration;         %% migration rate
double aggro;             %% aggressiveness for binary tournament select
string input;             %% input file containing FTSE-100 info
string output;            %% output file for statistics

randomize;                %% randomize the random number generators
read-data &input;         %% read the FTSE-100 info from the input file

%% generate a completely random starting population
doi &counter 1;
   random-genome;
   bqpd 1;
   assign-pop &counter;
untili &counter &popsize 1;

cycle

   %% place any immigrants on the stack
   immigrate &migration &processors &immigrants;

   %% evaluate them
   bqpd &immigrants;

   %% replace the worst members of the population with the immigrants
   replace-real-worst &immigrants &minimize;

   %% generate &popsize children using the following method
   doi &counter 1;

      %% select two parents
      binary-tournament &aggro &minimize;
      binary-tournament &aggro &minimize;

      %% perform crossover, mutation and evaluation operations
      rar &order;
      mutate &mutation;
      bqpd 1;

      %% replace the worst member of the population with the new child
      replace-real-worst 1 &minimize;

   untili &counter &popsize 1;

   %% rank the population to gain a median value
   rank-population &minimize;

   %% select the best member of the population for analysis
   select-best &minimize;

   %% make a clone of the best member, and migrate it to another
   %% slave process
```

```
      clone;
      migrate;

      analysis

         %% evaluate the genomes sent by the slave processes, and store
         %% them in the population
         bqpd &processors;
         doi &counter 1;
            assign-pop &counter;
         untili &counter &processors 1;

         %% rank the population to get a median value
         rank-population &minimize;

         %% every &period generations, save the statistics to the output file
         get-stats &output &period &minimize;

      endanalysis

   endcycle

   %% select the best solution for the final analysis phase
   select-best &minimize;

   analysis

      %% evaluate and store the genomes sent by the slave processes
      bqpd &processors;
      doi &counter 1;
         assign-pop &counter;
      untili &counter &processors 1;

      %% select the best one found by any slave
      select-best &minimize;

      %% save it to the output file
      save-genome &output;

   endanalysis

endplan

set &counter 0
set &processors 16
set &minimize -1
set &order 2
set &mutation 0.01
set &migration 0.05
set &aggro 0.6
set &immigrants 0
set &input 'FTSE-100'
set &output 'migrate.out'

set &popsize 50
set &cachesize 0
set &stacksize &processors

set &generations 1000
set &period 50

go &generations
```

B Input File Format

The input file format is very strictly defined. The first item must be an integer giving the size of the universal set. The second must be an integer giving the maximum subset size. Further items give the covariance matrix, market returns, mnemonics, full names and sectors. These must follow the following format.

Mnemonic A shortened name for the company.

Market Return The current market return for the company.

Full Name The full name of the company.

Sector The name of the sector that the company belongs to.

Covariance Matrix Entries The lower-left half of the matrix is stored, so if the company is the first in the list, this should consist of a single floating point number, if it is the 30th in the list, then this should consist of 30 floating point numbers.

An extract of the data for the FTSE-100 index follows. Note that there are in fact 104 companies included in the index.

```
104
30
ABF      1805.0 Ass_Brit_Food  Food
             234.91309
ABYNT    3500.4 Abbey_Natl Banks
             101.74891    595.38306
ALLD     5013.5 Allied_Lyons  Drinks
              57.86681    227.18004    457.60666
ARGY     3749.8 Argyll Retail
              94.60017    196.16054    143.05896    400.82883
ASSD     557.5 Asda Retail
             177.40991    338.45325    253.76328    334.96884   2998.01855
   .
   .
   .
   .
   .
```

References

[Goldberg, 1989] David E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. *Addison-Wesley*, 1989.

[Holland, 1975] John Holland. Adaption in Natural and Artificial Systems. *University of Michigan Press*, 1975.

[Kwiatkowski, 1991] Jan W. Kwiatkowski. Algorithms for Index Tracking. *Department of Business Studies, The University of Edinburgh*, 1991.

[Radcliffe, 1991] Nicholas J. Radcliffe. Forma Analysis and Random Respectful Recombination. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 222–229. Morgan Kaufmann (San Mateo, CA), 1991.

[Radcliffe, 1992] Nicholas J. Radcliffe. Genetic Set Recombination. In Whitley, editor, *Foundations of Genetic Algorithms 2*. Morgan Kaufmann (San Mateo, CA), 1992.

[Russo, 1991] Claudio V. Russo. A General Framework for Implementing Genetic Algorithms. Technical Report EPCC-SS91-17, Edinburgh Parallel Computing Centre, September 1991.

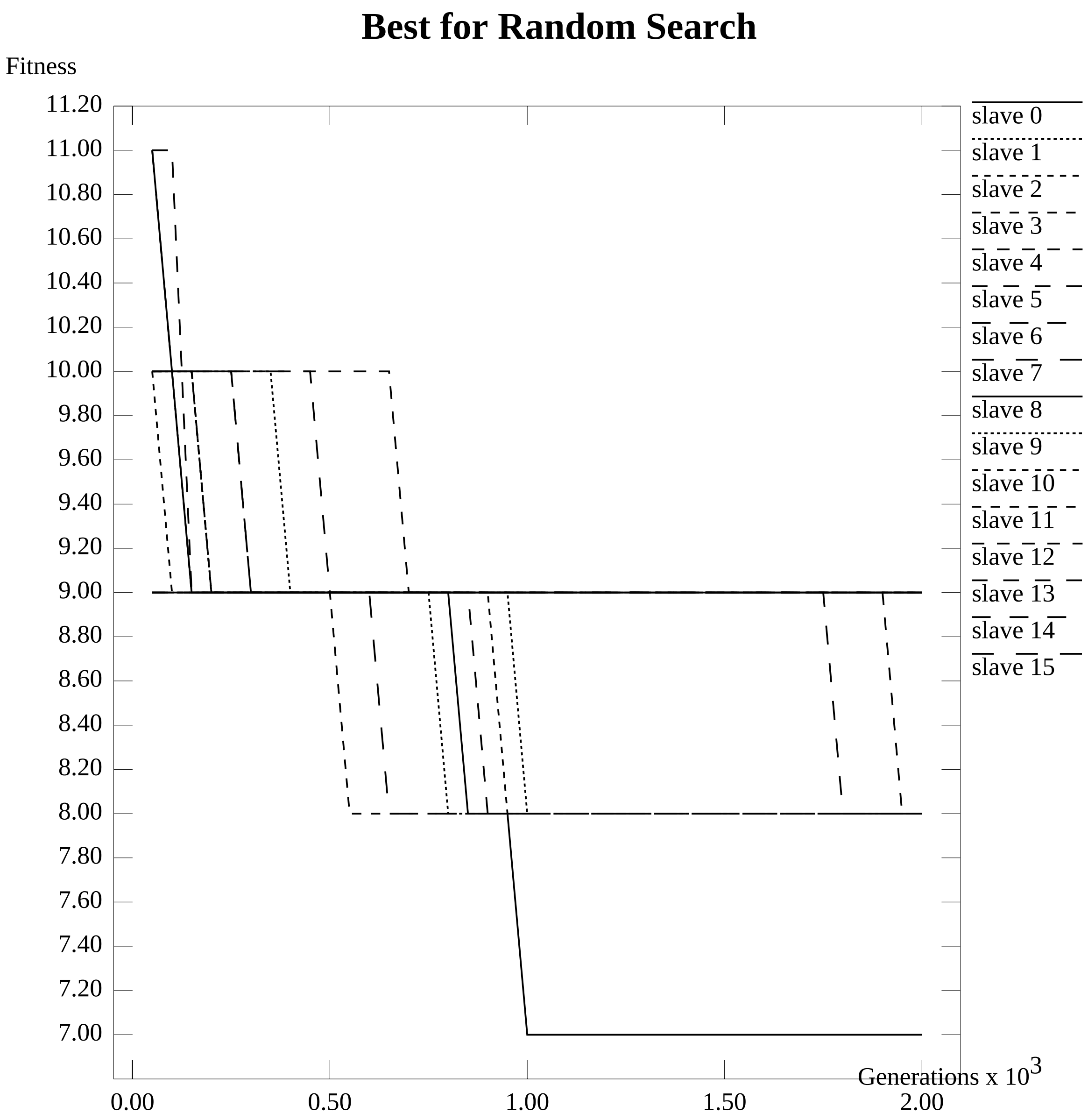

Figure 1: Best solutions attained for the simple objective function using random search

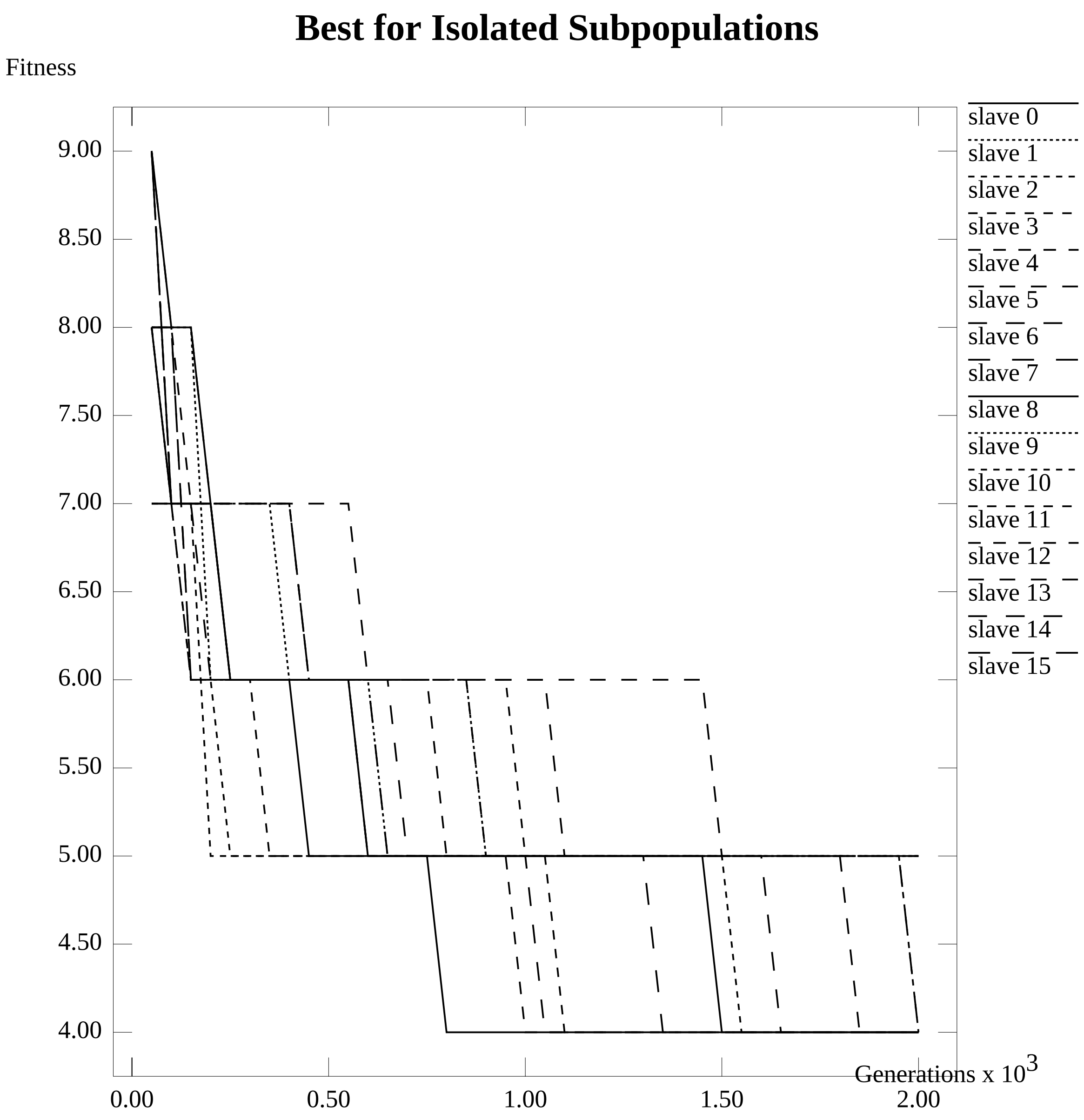

Figure 2: Best solutions attained for the simple objective function using a genetic algorithm with isolated subpopulations

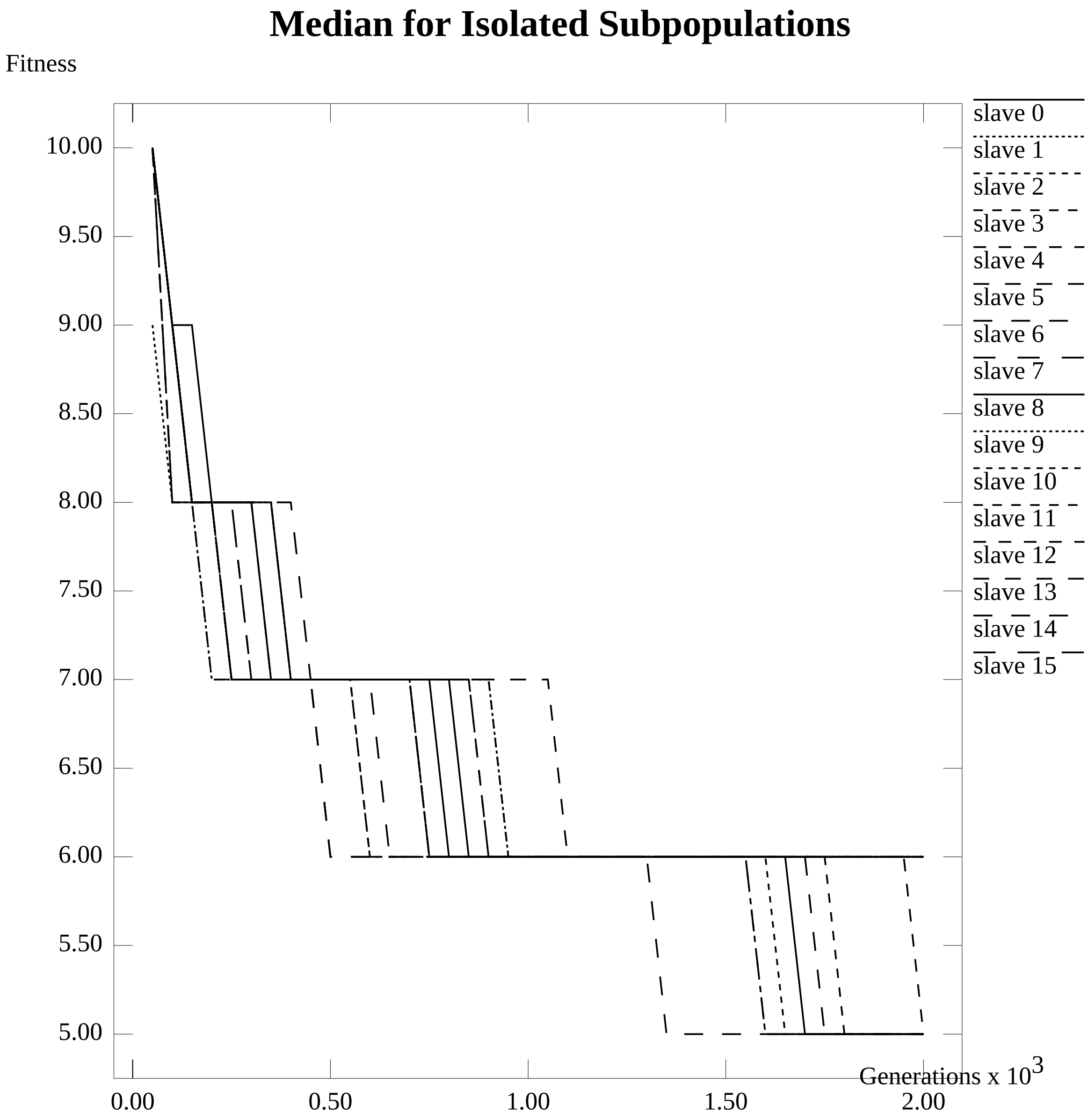

Figure 3: Overall view of the solutions attained for the simple objective function using a genetic algorithm with isolated subpopulations

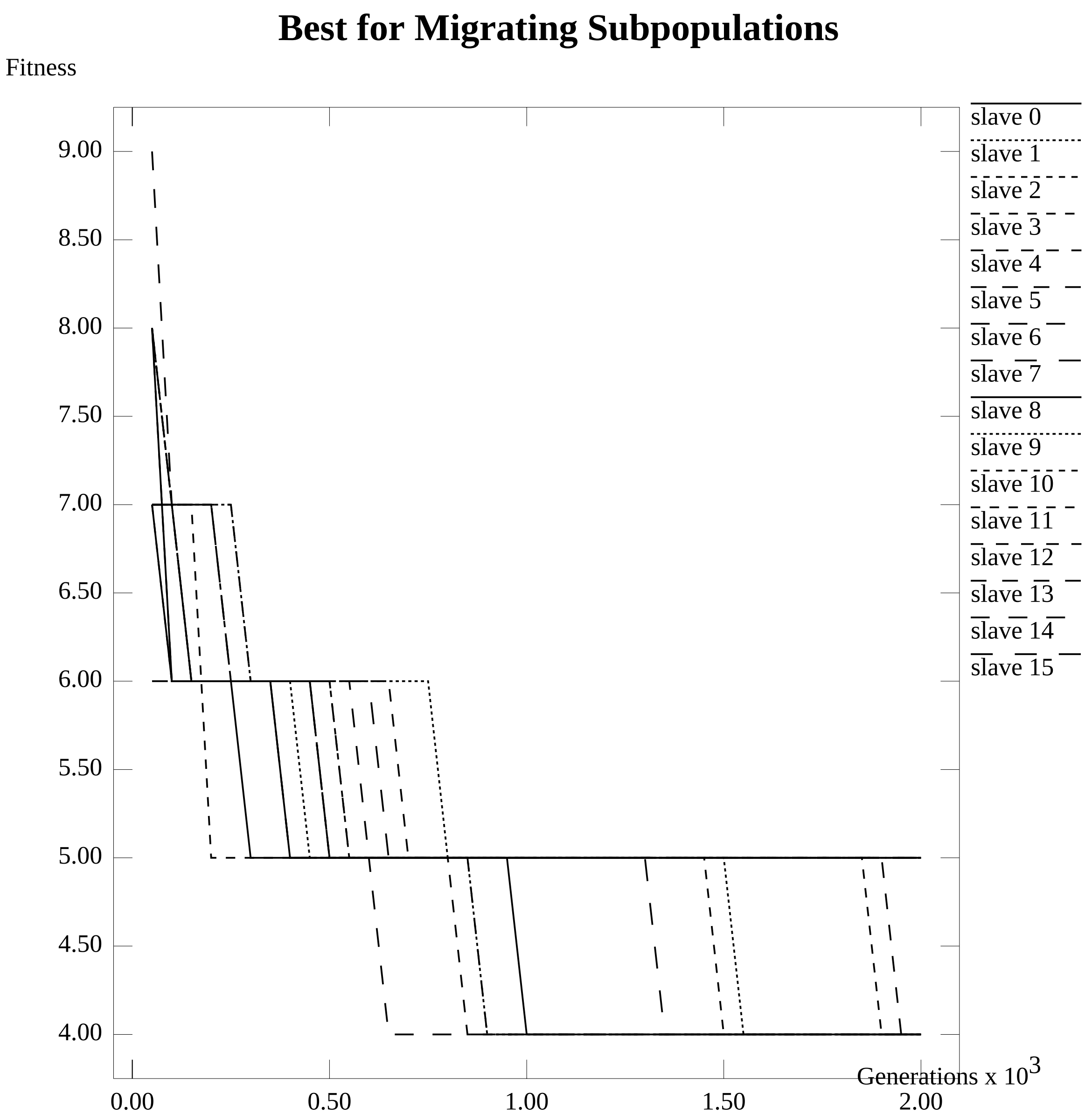

Figure 4: Best solutions attained for the simple objective function using a genetic algorithm with migrating subpopulations

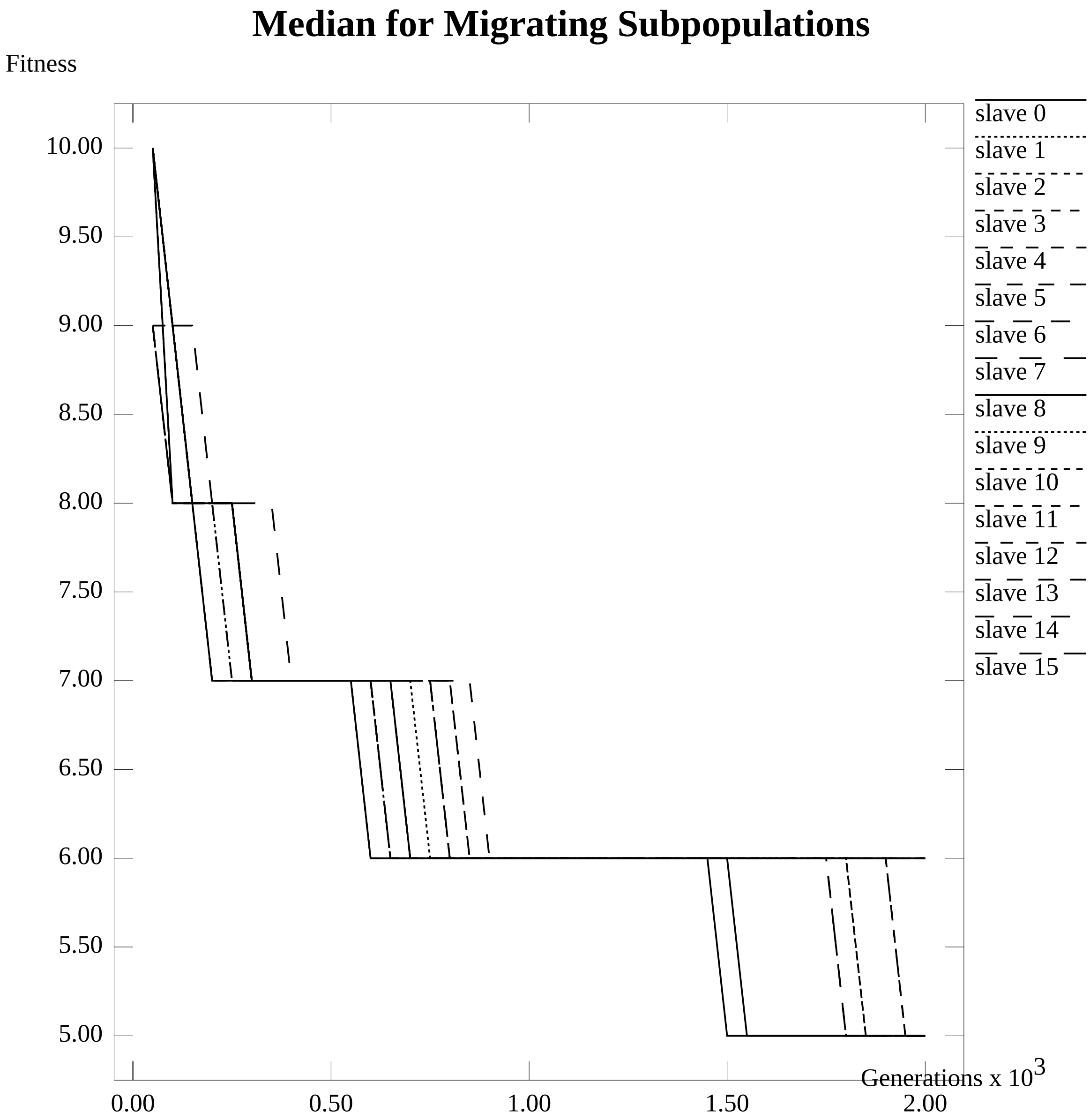

Figure 5: Overall view of solutions attained for the simple objective function using a genetic algorithm with migrating subpopulations

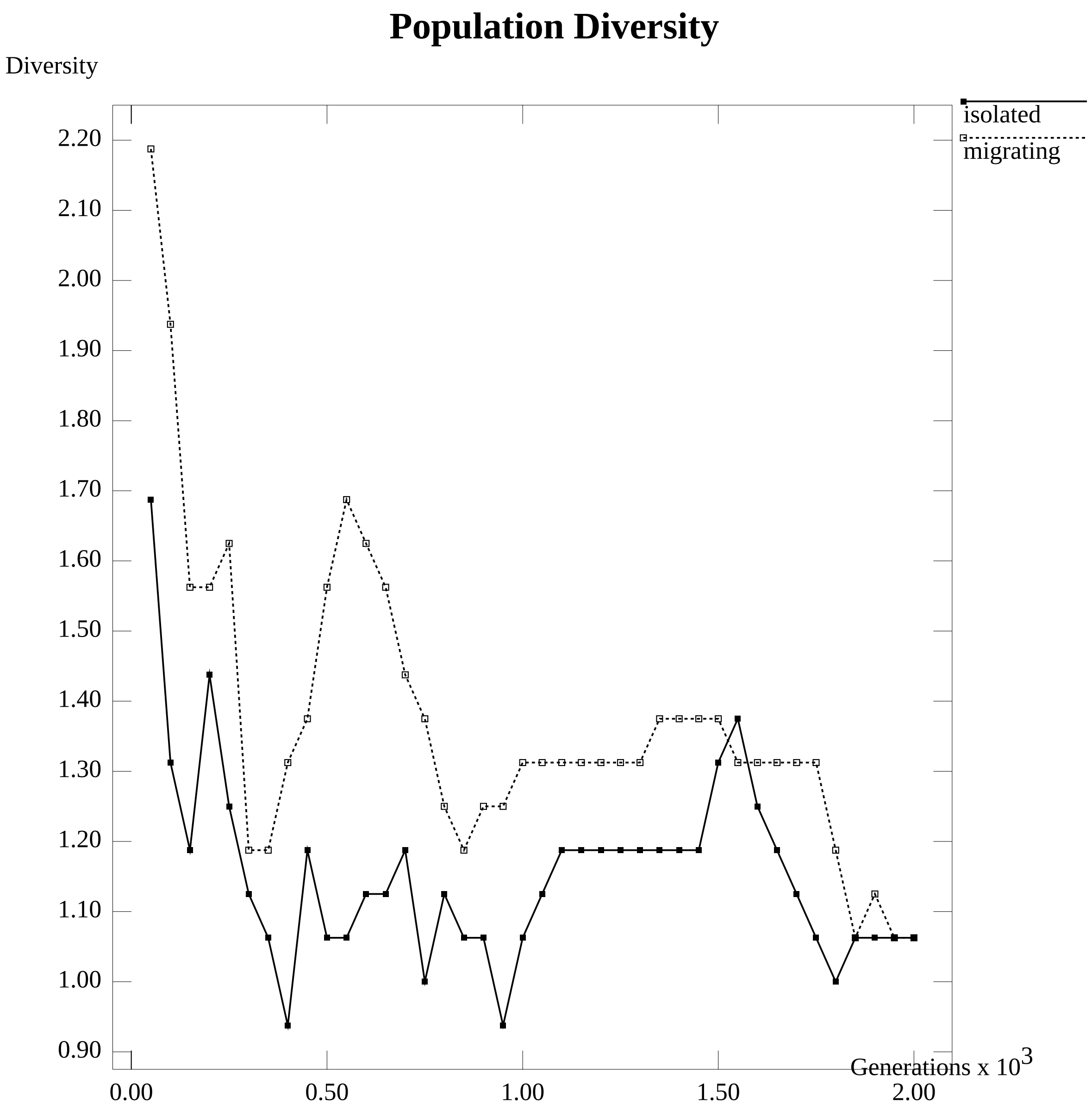

Figure 6: Population diversity obtained for the simple objective function using the two genetic algorithms

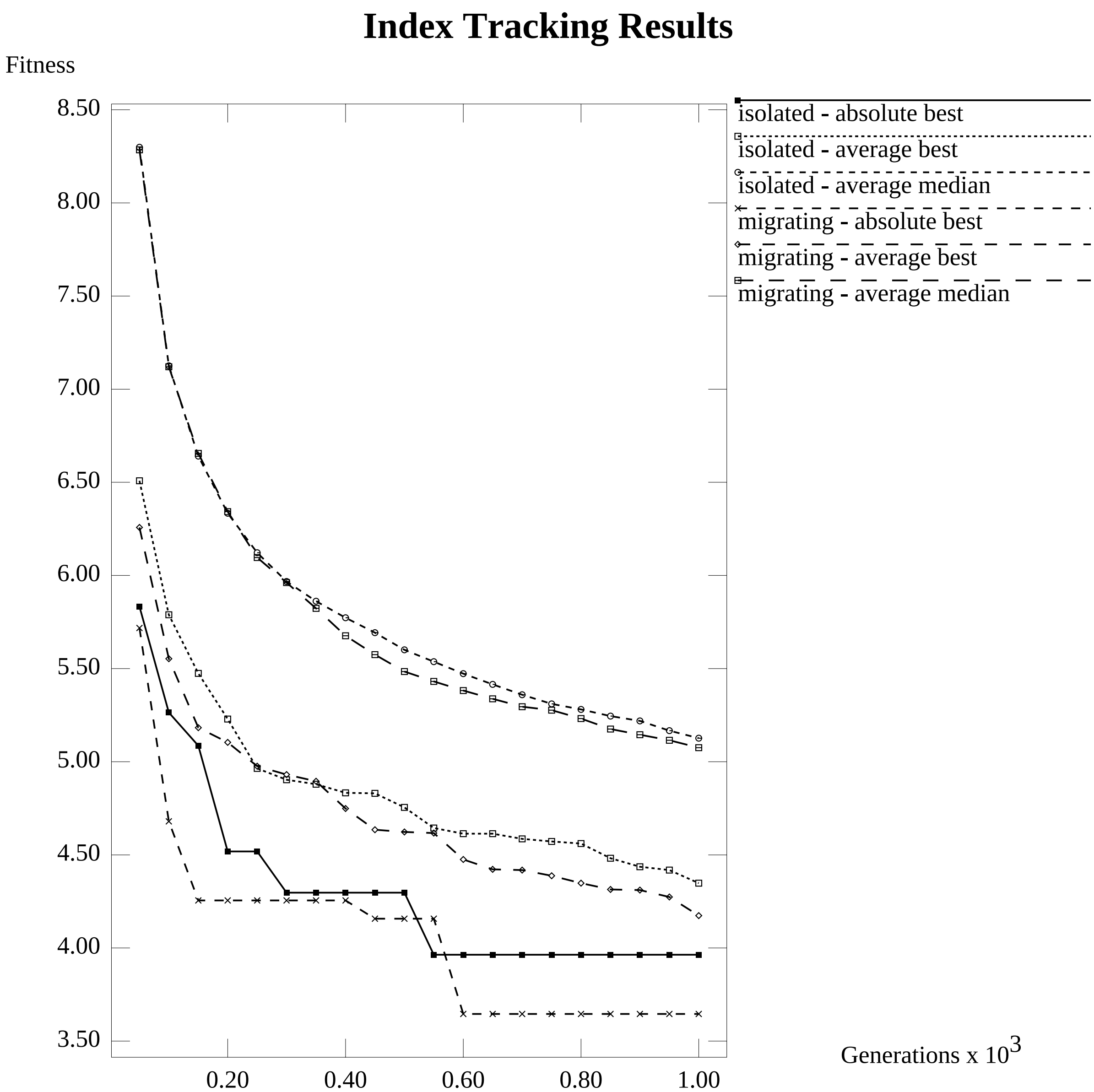

Figure 7: Results attained for index tracking using the two genetic algorithms

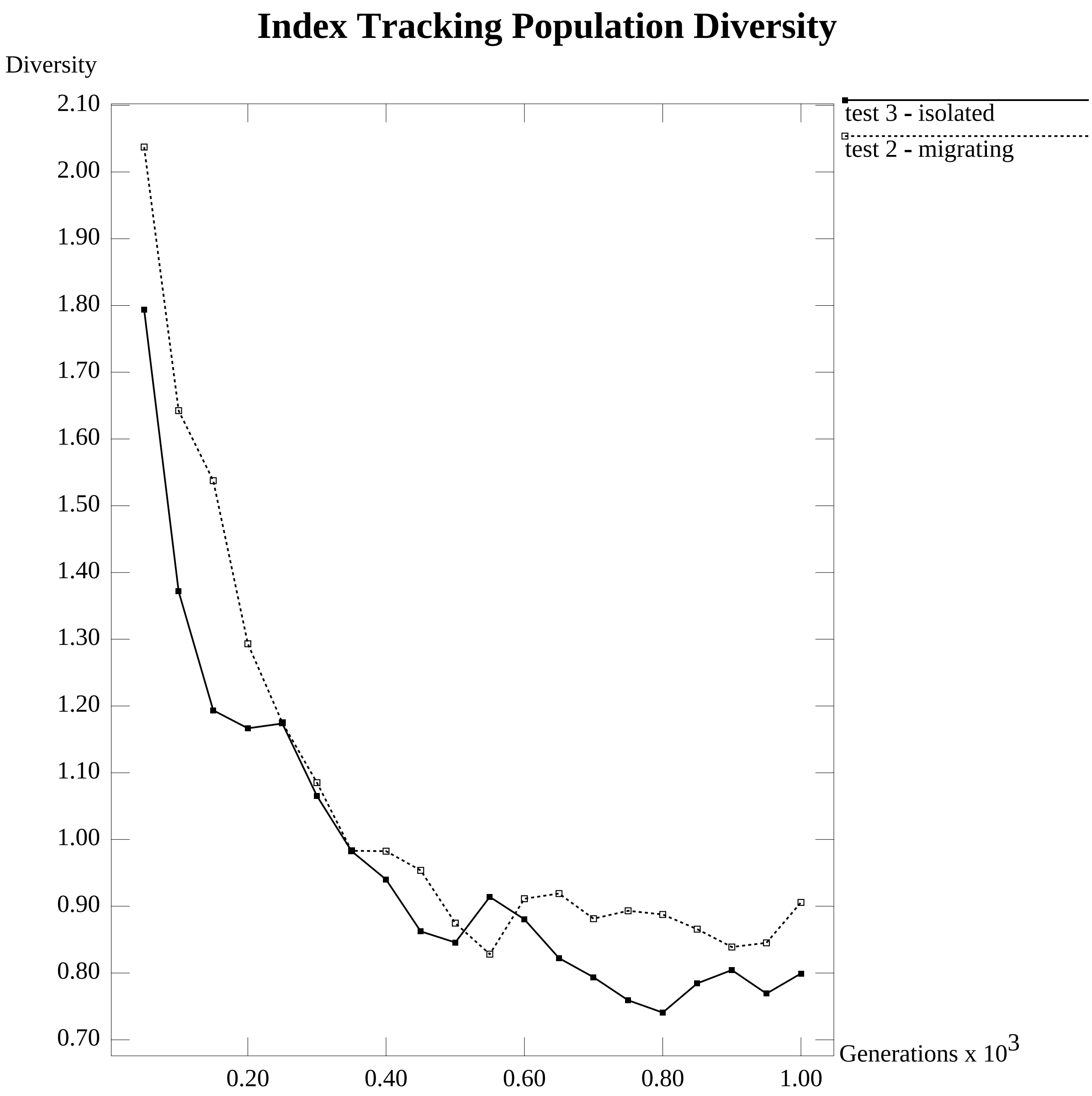

Figure 8: Population diversity obtained for index tracking using the two genetic algorithms