Genetic Algorithms for Use in Financial Problems

Andrew Jackson BCom(Hons) FIAA

Lend Lease Investment Management
Level 43 Australia Square Tower Building
Sydney NSW 2000 Australia

Telephone: +61 2 9237 5775
Facsimile: +61 2 9232 1850
e-mail: andrew_jackson@lendlease.com.au

Abstract

Genetic Algorithms apply the concepts of evolution to the solving of mathematical problems. This idea was first exploited by J.Holland in 1975 and has been applied to areas such as engineering, computing, biology and music. This paper will outline the basics of the genetic algorithm, and will apply the genetic algorithm approach to the problem of *asset allocation*, firstly using the traditional mean variance approach and secondly using a direct utility maximisation method for a step utility function. We will compare the performance of genetic algorithms with an alternative method of optimisation, Newton's method.

Keywords

Genetic Algorithms, Asset Allocation

Section I - What is a Genetic Algorithm?

Genetic algorithms are search procedures modelled on the mechanics of natural selection and evolution. They use techniques found in nature, such as reproduction, gene crossover and mutation to find optimal solutions to mathematical problems.

Genetic algorithms are most useful for problems with a large irregular search space where a global optimum is required. Traditional gradient based methods of optimistation encounter problems when the search space is multimodal (see Fig 1) as they tend to become stuck at local maxima rather than the global maxima. Genetic algorithms tend to suffer less from this problem of premature convergence.

Genetic algorithms may also be used for optimisation problems with a small well behaved search space (eg convex quadratic spaces (see Fig 2) as found in the traditional mean variance asset allocation type problem) but their performance (measured by speed and precision) relative to other more specialised algorithms, as demonstrated in this paper, will be poorer.

Fig 1

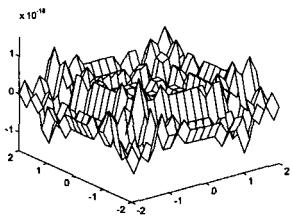
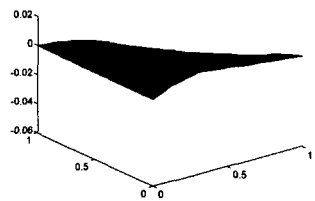
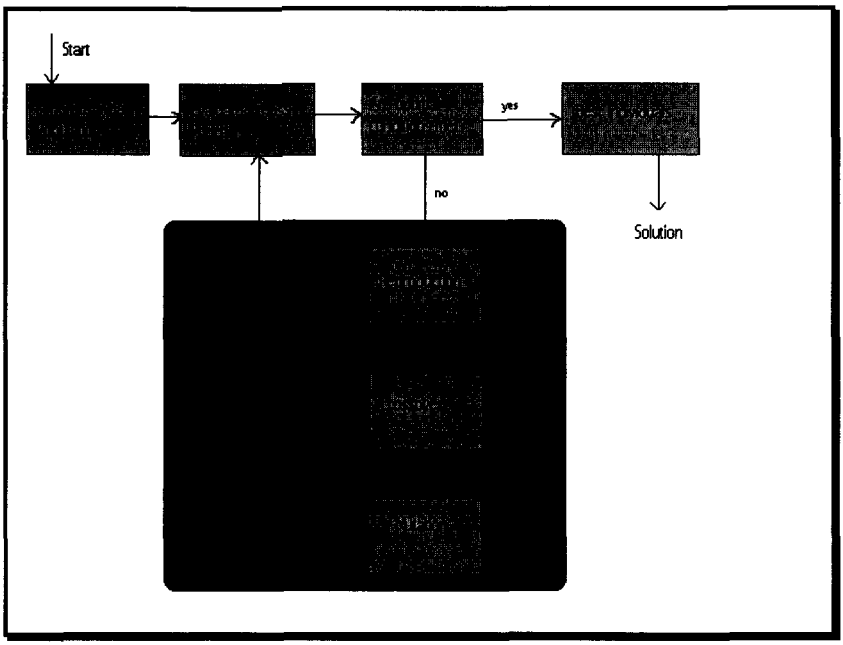


Fig 2



A genetic algorithm is an iterative approach, involving intelligent trial and error, which aims to find a global optimum. Nature's equivalent is the process of evolution over time, where many members are created, and each population becomes better and better adapted to its environment.

The basic structure of the algorithm is illustrated in the following flow chart[1],



During the procedure the genetic algorithm maintains a constant population size, P, of possible solutions (or members). In each iterative step, called a *generation*, the fitness of each member is evaluated and on the basis of this fitness, a new population of possible solutions is formed. Fit members are selected for reproduction, they find a mate in the population and exchange chromosome strings to create two new members (the children). To maintain diversity in the population (and prevent premature convergence to local optima) occasionally a mutated child is created. If this child has poor fitness, this mutation will quickly disappear from the population, but if it possesses high relative fitness it will spread through future generations. Once the children are created, the parents die, leaving the children as the new generation (or next step in the iteration). The process of reproduction, crossover and mutation is then repeated and future generations evolve. It is easy to see the similarity between nature (evolution) and the algorithm.

We will now go through the steps of the algorithm. These steps are outlined in many sources including Goldberg (1989), Grefenstette (1986) and many internet pages as cited in the bibliography.

1.1.1 Coding the problem as a finite length string

We must first code the problem. This involves putting the possible search space into a form that the genetic algorithm can work with. There are literally thousands of ways that any given problem could be coded into a finite length.
The most popular method is the *binary method.* This codes the problems variables into a binary string. For example, in a one dimensional search space x can take on the integer values 0 to 15. This search space could be coded as a 4 bit binary number eg 1001, 0111 etc, giving 15 combinations that can represent all possible values of x.

For a general problem, to code the variable V we,

1. Specify a range for the variable $[V_{min}, V_{max}]$ eg 0.2 to 0.8 (for the equity weighting in a portfolio)
2. Specify a length for the string , k , eg 8 bits or 16 bits
3. Map the parameter linearly from $[0, 2^k]$ to the range $[V_{min}, V_{max}]$.

This method allows us to specify the range of the variables (through $[V_{min}, V_{max}]$) and also the precision with which we want the parameter optimised (through the choice of the length of string). A larger range for the variable will result in less precision than a shorter interval. Also, a shorter string will result in lower precision than a longer string, but with the longer string leading to more computation time.

To maximise precision and minimise computation time, we should minimise the range of values that the variable should take and we should select a string length appropriate to our needs (for example, if our inputs are highly uncertain then it would be spurious to want a high level of precision for our optimised result).

Another method of coding is the *real number representation.* In this approach each variable being optimised is represented as a conventional floating point number and a one gene/one variable relationship is present. Changes must be made to the crossover and mutation operators in order to implement this approach. Mutation is achieved by adding a Gaussian-distributed random deviate scaled by a step size to the floating point value for each variable (unlike the mutation process described below).

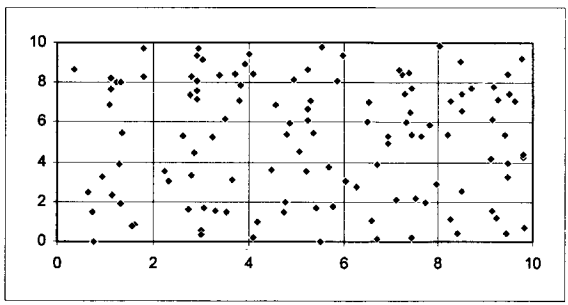
The method of coding can have a significant effect on the performance of the genetic algorithm.

1.1.2 Choosing an initial population P(0)

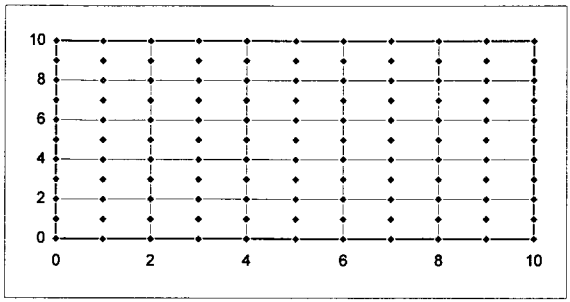
The most important aspect of choosing the initial population is deciding the *population size* to be used for the problem. This is the number of members contained in each generation. The population size has a significant effect on the performance of the algorithm. Genetic algorithms perform poorly if the population size is too small as not enough points in the search space are sampled leading to possible premature convergence to local maxima. A large population will overcome this problem, as a larger sample is evaluated. However, the larger population will require more fitness evaluations per generation. This can greatly increase the computing time, and lead to extremely slow speed of convergence. There is a trade off between convergence time and risk of premature convergence.

Typical ranges for populations are from 20-200. In general, a small, simple search space will require a smaller population. A 'standard' population size for most problems would be 50 members.

Once a population size n is decided then the initial population P(0) is chosen. P(0) can be chosen either randomly or methodically. With the random method, individuals are created from random guesses on the search space. For example in a 2 dimensional search space,



A methodical approach may space individuals evenly over the search space, for example,



The methodical method can also allow the user to concentrate the population in certain areas of the search space based on past experience of the likely optimal value.
Generally it is found that the method of choosing the initial population has little effect. The methodical approach will reduce variance (more efficient), but involves bias (as the user specifies the values chosen) while the randomised method is unbiased but may be less efficient.

1.1.3 Generate Fitness of Initial Population

For the genetic algorithm to work, we need some way of ranking individuals in order of fitness. This is achieved by the fitness function. *The fitness function varies depending on the individual problem.*
For example,

- If we are choosing x to optimise a quadratic function $f(x) = 2 + 2x - x^2$ then the fitness function is simply f(x).
- If we are choosing w1, w2, w3 (weights of assets in a portfolio) to maximize expected returns and minimize variance of a portfolio of assets then the fitness function may

look like $f(x) = 2t\mu - \sigma^2$, where μ is the expected return, σ^2 is the variance and t is a risk tolerance parameter.

Fitness functions should be ***strictly positive*** for all possible values. This can be achieved by a simple linear rescaling of the function eg $f^*(x) = f(x) - f(x_{\min}) + 1$.

A common feature in fitness functions is the use of ***penalty constraints*** which allow us to incorporate hard and soft constraints imposed on the optimisation problem. Including a penalty function transforms a constrained optimisation problem into an unconstrained optimisation problem on which the genetic algorithm can operate. For example if a superannuation fund wants to choose an optimal portfolio of assets from the asset classes cash, fixed interest and equity subject to the constraint that the fund may not hold more than 30% in cash. The fitness function may look like,

$$f(x) = \frac{2t\mu - \sigma^2}{\psi} \quad , \quad \text{where } \psi = \begin{cases} 1 & cash \le 30\% \\ k & cash > 30\% \end{cases}$$

where k is a constant, $k > 1$. This addition will decrease the fitness of any allocation which has over 30% in cash. If this is a hard constraint, we can set k to a very large number so members with over 30% cash have a very low fitness and are highly unlikely to be chosen for reproduction. If this is a soft constraint, we can adjust k to a suitable value (depending on the preferences of the trustees).

Specifying the fitness function is one of the primary tasks that a user of genetic algorithms must do as it determines what we are trying to optimise.

1.1.4 Generate P(t+1) from P(t)

For the basic genetic algorithm, generating the next generation of members involves three steps,

- selection
- crossover
- mutation

These are outlined below.

a) Selection

This is the process by which the members of the current generation are chosen for reproduction. Choosing members for reproduction can be done by a number of methods, the simplest being 'roulette wheel selection' where members are randomly chosen with probability proportional to their fitness.
For example,

Member	Fitness	Probability of Being Chosen
A	200	45%
B	100	23%
C	75	17%
D	55	13%
E	10	2%
Total	440	100%

Member A is most likely to be selected for reproduction, with member E being least likely.

Since we require the population size to be constant, and each mating produces two children, we must sample from the population n/2 times, where n is the population size and we are sampling with replacement. At each sample, any member has probability of being chosen in proportion to their relative fitness values. It is possible for a member to be mated with itself. In this example, the possible combinations of mates and the associated probabilities of a pair would be as follows,

Pair	Prob	Pair	Prob	Pair	Prob
AA	20.3%	BB	5.3%	CD	2.2%
AB	10.4%	BC	3.9%	CE	0.3%
AC	7.7%	BD	3.0%	DD	1.7%
AD	5.9%	BE	0.5%	DE	0.3%
AE	0.9%	CC	2.9%	EE	0.0%

Once all n/2 pairs of members are formed by a random sampling process, crossover occurs.

An alternative to roulette wheel (proportional) selection is *rank based selection.* This can reduce the scaling problems resulting from fitness functions. *Stagnation* can occur in proportional fitness if fitness values are very similar, with very low selective pressure being applied. Rank based selection can help overcome this problem. Ranking introduces a uniform scaling across the population and provides a simple and effective way of controlling selective pressure. As such, rank based fitness is generally more robust than proportional based fitness. Unfortunately, most codings for genetic algorithms use proportional fitness as it is simpler to implement.

b) Crossover

Genetic material from both the parents are combined to create two children.

The crossover procedure involves first *selecting a random point on the gene string.* The genes of both parents are then cut at this point and each half swapped to form two new children.

For example, if the parents are,

0100101 and 1100010

we must first select a position at which to cut. There are 6 possible positions in a 7 bit member. Say position 3 is randomly selected.

Parent 1 010|0101

Parent 2 110|0010

The two cut halves are swapped to form the children,

Child 1 010|0010

Child 2 110|0101

This process is repeated for all n/2 pairs of members and n new children are created.

We may also incorporate a feature called *elitism* where the fittest member of the current generation survives into the next generation. This means it is possible to have an immortal members in the population. Elitism is useful for later generations in the process as it concentrates the optimisation process around the best solutions[2].

Reproduction and crossover lead to *selection bias* in the population. Fitter members are more likely to reproduce and will hence come to dominate the population over subsequent generations. Reproduction and crossover are a powerful way of exploiting existing information in the population. Two fit parents will often combine to produce an even fitter child.

However, crossover and reproduction do not allow diversity in the population. New points on the search space are not examined and the process is in danger of converging

quickly to local (rather than global) maxima. To avoid this problem we introduce mutation.

c) Mutation

Mutation allows new information to be included in the population. This involves making infrequent random changes to the structure of the children produced. Each bit in every new child has the potential to mutate with the given probability. For example if the mutation rate is 1% then each bit in the new child 1100101 has a probability of 0.01 of changing in value from 0 to 1 or from 1 to 0.

Mutation protects against premature convergence to local maxima and is a random search across the whole solution space. Most mutations will quickly die out, while the few mutations that produce valuable new information will be incorporated into the population.

Mutation prevents a single bit becoming stagnant (or fixed for all members in a population). If the level of mutation is too low, there is a danger of premature convergence. If the level of mutation is too high, we are losing a lot of potentially valuable genetic information and the performance of the algorithm will fall. A very high level of mutation (>0.1) transforms the algorithm into a random search method (which is very inefficient). Grefenstette (1986) finds that high mutation rates greatly reduce the performance of the algorithm, as do extremely low mutation rates.
Typical mutation rates vary from 0.005 to 0.02. Poorly behaved search spaces may require higher mutation rates to prevent premature convergence.

1.1.5 Continue the process

After these operations (reproduction, crossover and mutation) have been carried out, we obtain a new generation P(t+1). The algorithm now enters another iteration, with the fitness of this new population being evaluated, then reproduction, crossover and mutation occurring, and the process continues until a satisfactory level of convergence has been achieved. Some stopping condition must be specified.

Most software packages simply require the user to specify the fitness function for the problem and the range of allowable values for the parameters, so users do not need to be overly concerned with the process of reproduction, crossover and mutation to obtain a suitable solution.

The steps of reproduction, crossover and mutation form the basis of the genetic algorithm approach. Genetic algorithms differ from traditional search techniques in a number of ways[3],

1. Genetic Algorithms are good at balancing the trade off between exploring new points in the search space and exploiting information already discovered.
2. Implicit parallelism is involved as a number of solutions are worked on simultaneously improving efficiency and reducing the chance of premature convergence to local maxima.
3. Genetic Algorithms are randomised, so results are determined by probability.
4. They do not use gradients or any other form of derivatives to find solutions.
5. The approach involves a coding of the parameter set, rather than working with the parameters themselves.

Advantages of Genetic Algorithms[4]

- They require no knowledge or gradient information about the search space
- Discontinuities present on the response surface have little effect on overall optimisation performance
- They are resistant to becoming trapped in local optima
- They perform very well for large-scale optimisation problems
- Can be employed for a wide variety of optimisation problems

Disadvantages of Genetic Algorithms

- Have trouble finding the exact global optimum
- Require large number of response (fitness) function evaluations
- Require a coding of the problem
- More efficient algorithms are available for many specific problems, especially when the problem is simple, small or mathematically well behaved (as we will see subsequently).

Section II - Practical Application of Genetic Algorithm

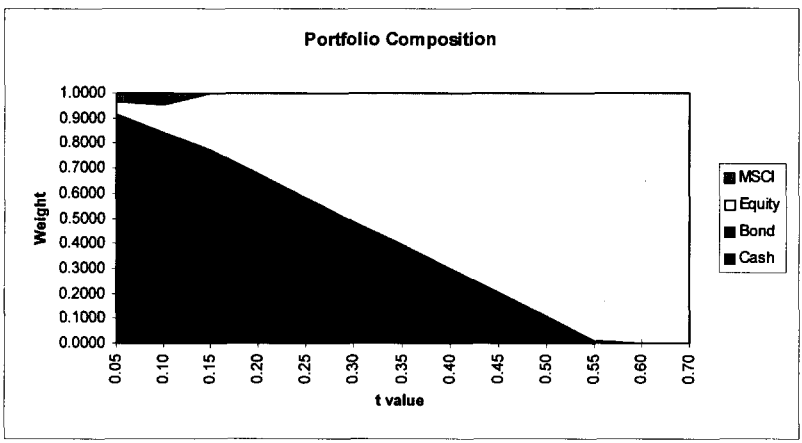
Example I - Mean Variance Asset Allocation

We will examine a simple application of using genetic algorithms in a financial problem. Consider a superannuation fund's asset allocation decision. Traditionally we could perform this using a mean variance optimisation as outlined in Merton (1972), this method gives us an exact solution to the problem of maximising return for minimum variance . We could also use numerical methods such as the genetic algorithm or Newton's method to choose the optimal asset mix for the fund.

Using Newton's method we get the following results,

t	*Cash*	*Bond*	*Equity*	*MSCI*	*E(R)*	*Var*	*St Dev*
0.05	0.4627	0.4549	0.0510	0.0314	0.0073	0.00009	0.00952
0.10	0	0.8422	0.1131	0.0447	0.0089	0.00031	0.01750
0.15	0	0.7738	0.2262	0	0.0091	0.00037	0.01920
0.20	0	0.6787	0.3213	0	0.0093	0.00044	0.02100
0.25	0	0.5836	0.4164	0	0.0095	0.00053	0.02311
0.30	0	0.4885	0.5115	0	0.0097	0.00065	0.02545
0.35	0	0.3935	0.6065	0	0.0099	0.00078	0.02796
0.40	0	0.2984	0.7016	0	0.0101	0.00094	0.03061
0.45	0	0.2033	0.7967	0	0.0104	0.00111	0.03336
0.50	0	0.1082	0.8918	0	0.0106	0.00131	0.03618
0.55	0	0.0132	0.9868	0	0.0108	0.00153	0.03906
0.60	0	0	1	0	0.0108	0.00156	0.03947
0.70	0	0	1	0	0.0108	0.00156	0.03947

Graphically, this is as follows,



First we must derive a fitness or objective function. We will assume that the fund is trying to maximise expected utility of wealth.

As outlined in Mueller (1996), expanding the utility function in a Taylor series around the expected end of period wealth u(E(W)) gives,

$$u(W) = u(E(W)) + u'(E(W))(W - E(W)) + \frac{1}{2}u''(E(W))(W - E(W))^2 + \frac{1}{6}u'''(E(W))(W - E(W))^3 + ...$$

Taking expected values gives,

$$E(u(W)) = u(E(W)) + u'(E(W))E(W - E(W)) + \frac{1}{2}u''(E(W))E(W - E(W))^2 + \frac{1}{6}u'''(E(W))E(W - E(W))^3 + ...$$

If we assume quadratic utility, $U(W) = aW - bW^2$, this reduces to,

$$E(u(W)) = u(E(W)) + \frac{1}{2}u''(E(W))\sigma^2(W)$$

where $u(E(W)) = E(aW - bW^2) = aE(W) - bE(W^2)$

$$u''(E(W)) = -2b$$

$$W = W_0(1 + R)$$

We can make a positive transformation to the utility function to give the objective function (ignoring the $[E(R)]^2$ term),

$$\boxed{E(u(W)) = 2tE(r) - \sigma^2}$$

where

$t = \dfrac{Wu'(E(W))}{u''(E(W))}$, the 'risk tolerance' of the super fund.

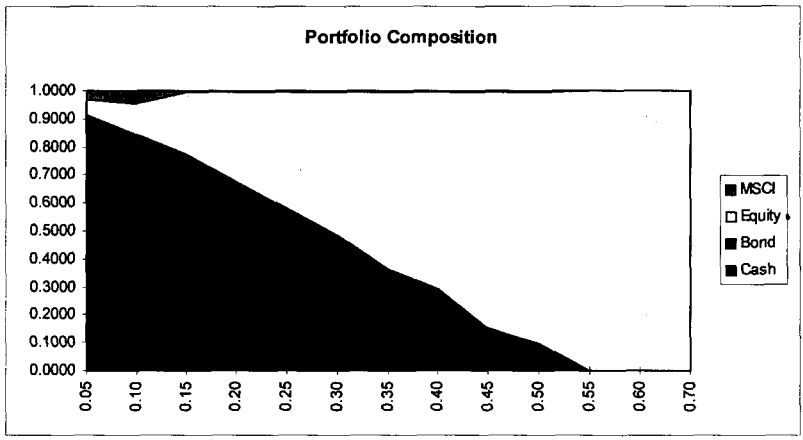
We will use the following return and volatility assumptions (obtained from Australian monthly data for the period June 1991 to May 1996),

Asset	E(r)	Standard Deviation	Variance
Cash	0.00563	0.00139	0.0000019
Bonds	0.00862	0.01672	0.0002800
Equity	0.01079	0.03947	0.0015580
MSCI	0.00853	0.03571	0.0012750
Correlation Matrix			
1	0.3820	0.1055	0.0087
	1	0.5259	0.2753
		1	0.4915
			1

Using the genetic algorithm (evaluating the optimum allocation for each t value) we get the following results,

t	*Cash*	*Bond*	*Equity*	*MSCI*	*E(r)*	*Var*	*St Dev*
0.05	0.4706	0.4431	0.0549	0.0314	0.0073	0.00009	0.00944
0.10	0	0.8431	0.1137	0.0431	0.0089	0.00031	0.01751
0.15	0	0.7725	0.2235	0	0.0091	0.00037	0.01915
0.20	0	0.6745	0.3216	0	0.0093	0.00044	0.02101
0.25	0	0.5804	0.4157	0	0.0095	0.00053	0.02310
0.30	0	0.4863	0.5098	0	0.0097	0.00065	0.02542
0.35	0	0.3647	0.6314	0	0.0100	0.00082	0.02867
0.40	0	0.2941	0.7020	0	0.0101	0.00094	0.03065
0.45	0	0.1529	0.8431	0	0.0105	0.00121	0.03476
0.50	0	0.0980	0.8980	0	0.0106	0.00132	0.03640
0.55	0	0	1	0	0.0108	0.00156	0.03947
0.60	0	0	1	0	0.0108	0.00156	0.03947
0.70	0	0	1	0	0.0108	0.00156	0.03947

Graphically, this is as follows,



We can see that the portfolio compositions are very similar for both methods. However, using Newton's method to calculate the optimal portfolios took around 5 seconds computation for each t value, whilst the genetic algorithm took around 2 mins for each t value. The performance of the genetic algorithm was poor relative to the specialised algorithm in this simple quadratic convex search space, and the results acheived were close to, but not equal to the exact optimal values (as given by Newton's method).

Example II - Step Utility Asset Allocation

In this case we will compare the Newton's method to the genetic algorithm if the fund has a step utility function (as outlined in Lipman (1989)) rather than a quadratic utility function.

We will assume there are a number of steps in the utility functions associated with various key levels of return for the fund. Let the utility function be as follows,

$$u(W_1) = u(W_0(1+R)) = \begin{Bmatrix} -e^{-cW_0(1+R)} \ldots\ldots\ldots\ldots R \geq B \\ -b_1 - e^{-cW_0(1+R)} \ldots\ldots 0 < R \leq B \\ -(b_1+b_2) - e^{-cW_0(1+R)} \ldots R < 0 \end{Bmatrix}$$

Utility falls by a step of size b_1 if we fail to meet the benchmark return level $B > 0$. Utility falls by a further step b_2 if we fail to achieve a positive return.

Expected utility is as follows,

$$E(u(W_1)) = -\int_B^{\infty} e^{-cW_0(1+R)} f(R)dR + \int_0^B \left[-b_1 - e^{-cW_0(1+R)}\right] f(R)dR + \int_{-\infty}^0 \left[-b_1 - b_2 - e^{-cW_0(1+R)}\right] f(R)dR$$

This can be written as,

$$E(u(W_1)) = -\int_{-\infty}^{\infty} e^{-cW_0(1+R)} f(R)dR - \int_0^B b_1 f(R)dR - \int_{-\infty}^0 (b_1+b_2) f(R)dR$$

If returns are normally distributed then this becomes,

$$-e^{-cW_0} e^{-\mu cW_0 + \frac{1}{2}\sigma^2 c^2 W_0^2} - b_1 \Phi\left(\frac{B-\mu}{\sigma}\right) - (b_1+b_2)\Phi\left(\frac{0-\mu}{\sigma}\right)$$

Note: If we generalise to an n-step utility function then the fitness function will be of a similar form with n penalty terms.

For this example, we will assume that the benchmark return for the fund is 0.6% per month (just higher than the average cash return). It is possible to use a variable benchmark rate for this problem (such as the cash rate or the median managers return) but this greatly complicates the structure of the objective function.

We will set the relative risk aversion at 0.2 (close to the Australian average level for managed funds implied by their holdings).
Australian monthly returns over the period June 1991 to May 1996 will be used.

Different values of b_1, b_2 will be used to illustrate the effect of varying step sizes.

The objective function for the problem can be written as,

$$F = -e^{-5}e^{-\mu 5+\frac{1}{2}\sigma^2 25} - b_1\Phi\left(\frac{0.006-\mu}{\sigma}\right) - (b_1+b_2)\Phi\left(\frac{0-\mu}{\sigma}\right)$$

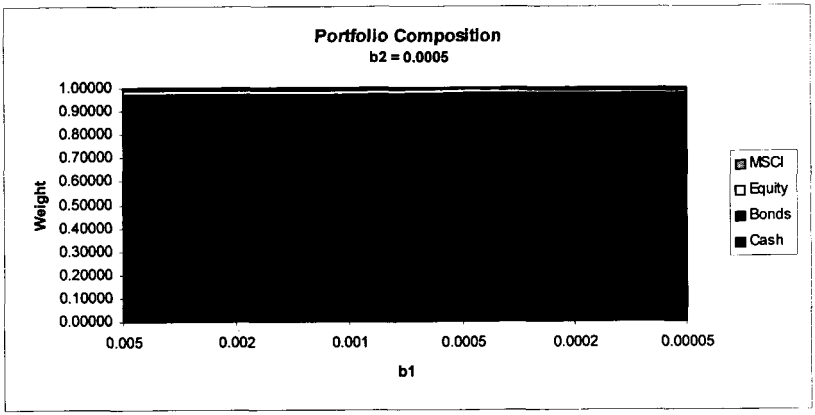
Not surprisingly, Newton's method is very unstable for this problem as a gradient based method would be expected to have difficulty with steps in the utility function. For given values of b_1, b_2 Newton's method produces different portfolio compositions depending on the starting values chosen. For example, setting $b_1 = 0.001$, $b_2 = 0.002$ Newton's method gives the following results for different starting values,

Starting Weights	1.00	0	0	0	0.50
	0	1.00	0	0	0.50
	0	0	1.00	0	0
	0	0	0	1.00	0
Optimised Weights	0.957	0.906	0.906	0.957	0.908
	0	0.094	0.094	0	0.070
	0.043	0	0	0.043	0.013
	0	0	0	0	0.009

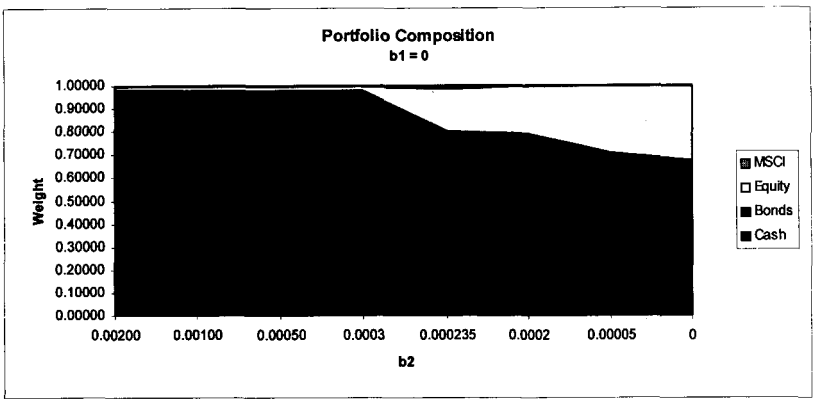
We can see that Newton's method gives three different optimised portfolios depending on the starting weight combination chosen.

The genetic algorithm does not rely on a particular starting value, so performs much better in this ill-conditioned problem.
The optimal portfolio results for different step values are as follow, each chart holds one step size fixed and varies the size of the other.

Varying b_1 gives the following results,



The portfolios are composed primarily of cash in the above case, as the step function discourages higher volatility portfolios. There is little variation as we vary the step size.



Past a certain step size, there is a sudden change in portfolio composition, from around 90% in cash to 0% in cash, with a small change in step size from 0.0003 to 0.0002. This represents a discontinuity in the search space.

The genetic algorithm took around 1 minute to evaluate each step size combination, this is still reasonably slow relative to Newton's method, but it does not suffer from premature convergence. The speed differential with respect to Newton's method will diminish as we increase the size and complexity of the problem.

Conclusion

The genetic algorithm is a viable alternative optimisation method to Newton's method. For well conditioned problems the Newton algorithm will be faster and more precise. However as the complexity of the problem increases, the relative efficiency of the genetic algorithm will improve. The genetic algorithm is more robust to discontinuities in the search space, and is not as sensitive to starting values as is Newton's method.

Genetic algorithms can be used to solve the asset allocation problem. All we must do is specify an objective function for the fund (such as max expected utility), the constraints on the asset weights, and any penalty functions (such as utility steps) and the genetic algorithm will find the optimum asset allocation for the fund. We can also use general risk measures (such as semivariance or probability of shortfall) and general objective functions (any form of utility function) without greatly effecting the performance of the algorithm.

Acknowledgements

Les Balzer Lend Lease Investment Management, Sydney

Mike Sherris Macquarie University, Sydney

Bibliography

Buckles and Petry (1992), *Genetic Algorithms*, IEEE Computer Society Press

Ward Systems Group (1996), *Users manual for GeneHunter software.*

Grefenstette (1986), 'Optimisation of Control Parameters for Genetic Algorithms', *IEEE Transactions on Systems, Man and Cybernetics*, 16, Number 1, 1986, pp 122-128.

Jackson, AR. (1996), 'Genetic Algorithms and Asset Allocation', Unpublished Honours Thesis, Macquarie University Australia.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley.

Lipman, R A (1989), 'Utility, Benchmarks and Investment Objectives', *Transactions of the Institute of Actuaries of Australia.*

Markowitz, H (1959), *Portfolio Selection*, Blackwell : Yale.

Merton, R. (1972), 'An Analytic Derivation of the Efficient Portfolio Frontier', *Journal of Financial and Quantitative Analysis,* Sep 1972.

Meuller, H. (1996), *Financial Economics for Actuaries,* unpublished manuscript.

Internet Starting Points

Genetic Algorithms Archive, http://www.aic.nrl.navy.mil/galist/

Nova Genetica by Darin Molnar, http://www-adm.pdx.edu/user/anth/darin/daringa.htm

Newsgroup: comp.ai.genetic

---

[1] Modified from Hartmut Pohlheim, Daimler Benz AG, Research and Technology internet page. http://www.systemtechnik.tu-ilmenau.de/~pohlheim/

[2] GeneHunter Users Manual (1996)

[3] Buckles and Petry (1992), *Genetic Algorithms*, IEEE Computer Society Press

[4] The Genetic Algorithm Archive , http://chem1.nrl.navy.mil/~shaffer/practga.html