# ‘Simple’ differential evolution for beef model optimisation

D. G. Mayer[a], B. P. Kinghorn[b] and A. A. Archer[c]

[a]Queensland Department of Primary Industries, Brisbane. david.mayer@dpi.qld.gov.au

[b]Sygen Chair of Genetic Information Systems, University of New England, Armidale.

[c]CRC for Cattle and Beef Quality, University of New England, Armidale.

**Abstract:** Differential evolution (DE) is a comparatively simple variant of the broad class of evolutionary algorithms, which encompass genetic algorithms, evolution strategies, genetic programming and hybrids of these. DE has only three operational parameters, and can be coded in 20 lines of pseudo-C. Investigations of its performance in the optimisation of a challenging 70-dimensional beef property model indicate that it performs at least as well as Genial (a real-value genetic algorithm), which has been the preferred operational package thus far. Despite DE’s apparent simplicity, the interacting key evolutionary operators of mutation and recombination are present and appear to be effective. In addition, DE has the advantage of incorporating a form of self-adapting mutation, as found in evolution strategies, without the burdening overhead of doubling the dimensionality of the search-space. These processes are illustrated, and model optimisations totalling two years of Sun workstation computation are presented. These results show that the baseline DE parameters work effectively, but can be improved in two ways – firstly, the population size does not need to be overly conservative, and smaller populations can be considerably more efficient; and secondly, the periodic application of extrapolative mutation counteracts the contractive nature of DE’s intermediate arithmetic recombination in the latter stages of the optimisations. This provides an escape mechanism to prevent sub-optimal convergence. With its ease of implementation and proven efficiency, DE is ideally suited to both novice and experienced users wishing to optimise their simulation models.

***Keywords:*** *evolutionary algorithm, beef model, optimisation, differential evolution*

## 1. INTRODUCTION

Systems research and modelling have now become mainstream techniques in many fields, including agriculture. The modelling steps of system definition, development, programming, verification, validation, and model investigation have been well documented (Bratley et al., 1987), and these steps have repeatedly proven to be useful in themselves, to obtain an understanding of the dynamics of the system under study. Logically, these models can also be taken to the next step, which is formal optimisation. This task becomes increasingly difficult as the size and complexity of the model increases (Meadows and Robinson, 1985).

To conduct this optimisation, an objective function must first be constructed, with the optimisation algorithm then trialing various values of the input parameters to search for the best (optimal) combination. This objective function is usually taken as some overall economic measure of the system, for example the gross margin or profit of a farming enterprise or system. Non-economic measures can also be optimised (e.g. total farm production, or pesticide or nutrient runoff), however the optimal values of these will usually occur when using maximal (and prohibitively expensive) inputs. When there are potentially competing requirements in the simulated system (e.g., a requirement to maximise profit whilst simultaneously minimising soil loss), a compromise of these is usually considered via a weighted objective function. Alternately, one of the range of more complex Pareto multi-criteria evaluation methods (Coello Coello et al., 2002) can be used for the objective function to be optimised.

A large range of optimisation methods and algorithms can be found in the mathematical, computing, operations research and applied literature. Agricultural models generally pose some of the more difficult problems for these methods - these complex models cannot be numerically differentiated; the ‘curse of dimensionality’ often applies to give very large and complex search-spaces; they typically have non-smooth response surfaces (including sharp cliffs when the system is over-utilised, and collapses both biologically and economically); multiple local optima (in overall economic terms) can confound the search; and epistasis (the interacting effects of the input variables) is usually pronounced.

This range of problems means that only the most efficient optimisation methods are likely to succeed with 'real-world' scale agricultural models. In particular, the somewhat dated (but still widely used) multitude of gradient methods are poorly suited for this task. Similarly, deterministic direct-search methods (including the robust Simplex method, Nelder and Mead, 1965) also struggle here. Of the more modern stochastic algorithms, tabu search (Glover et al., 1993) and ant-colony methods are well suited to combinatorial optimisation problems, but not to the optimisation of multi-dimensional models. Similarly, simulated annealing (Kirkpatrick et al., 1983) has proven to be thorough and reliable, but is too inefficient to be of practical use with larger problems (Mayer, 2002). This leaves evolutionary algorithms as the only practical methodology.

Evolutionary algorithms encompass a range of different 'nature-inspired' methods, including genetic algorithms (usually binary representation, with recombination the primary operator), evolution strategies (real-value representation, with mutation the primary operator), and genetic programming (variable-length representation, more usually aimed at developing equations and programs). Whilst independently developed for a number of years, these sub-strains of evolutionary algorithms have now effectively merged, with each adopting the more favorable features of the others. Differential evolution (DE) is one such hybrid.

In keeping with the large range of potential operators (e.g., controlling the many types and rates of selection, crossover and mutation), most optimisation software has tended to be 'large'. This poses potential problems for users, in that they cannot be sure that these particular operators are correctly coded and actually doing what they are supposed to be doing. Table 1 lists four examples of algorithms which can be easily obtained.

DE is simpler to code, implement and use than other optimisation methods (Table 1). The following sections introduce its methodology, and put these into context in comparison with other evolutionary algorithms. A large, complex beef property model is then used as a case study for these methods, and general conclusions regarding relative efficiency and performance are drawn.

**Table 1.** Types and sizes of shareware optimisation routines (SA = simulated annealing, GA = genetic algorithm, ES = evolution strategy).

| Package | Type | Code | Lines |
|---|---|---|---|
| ASA[a] | (adaptive) SA | C | 8,806 |
| Genesis[b] | Binary GA | C | 2,829 |
| Genial[c] | Real-value GA or ES | Fortran | 3,532 |
| DE[d] | Real-value GA | C | 20 |

[a] http://www.ingber.com/#ASA
[b] http://www.aic.nrl.navy.mil/galist/src/
[c] http://hjem.get2net.dk/widell/genial.htm
[d] http://www.icsi.berkeley.edu/~storn/code.html

## 2. DIFFERENTIAL EVOLUTION

Details of DE, including the 20-line pseudo-C code, are listed in Storn and Price (1997) and Price and Storn (1997). On test functions, DE has markedly outperformed both simulated annealing and the Simplex method, and was equal or superior to some common evolutionary algorithms (Storn and Price, 1997). A number of rather complex versions of DE are available on the DE website (see address in Table 1 footnote), and for general use we have included a simplified and commented Fortran version in Appendix 1 of this paper.

The concept of DE is simple. Firstly, a population of candidate members (trial management strategies for the model) is established, usually at random. Each population member is characterized by its fitness (its value on the target objective function). For each population member in turn, a challenger is constructed. If this challenger has superior fitness, it will replace the population member in the next generation. To construct this challenger, three other population members are chosen at random. We can label these as *a*, *b* and *c*. Each parameter (management option, as coded on to DE's alleles) is then addressed in turn. With a probability equal to the crossover rate (CR), the parameter is simply adopted from the population member that the challenger is challenging. Otherwise, a new parameter value is constructed as the value for member *a* plus the mutation factor (F) times the difference between the values for *b* and *c*. Successful challengers replace their respective population members, and, together with surviving members, constitute a new generation with higher mean fitness. The process continues over sufficient generations to achieve convergence close to an optimal solution, with the fittest solution being chosen.

One possible reason that DE works so well is that mutation is driven by differences between parameter values of contemporary population members, giving an appropriate reduction in magnitude as the optimisation proceeds and convergence is approached. This parallels the successful approach used in evolution strategies, where the mutation variances are self-tuning. To achieve this feature the evolution strategies take these standard deviations or variances (one per parameter being optimised) along as extra parameters to be optimised. This effectively doubles the dimensionality, and hence the search-space, of each problem. DE's approximate approach does not require this doubling of the problem's size, and thus appears to be a far more efficient implementation of self-adapting mutation.

## 3. OPERATIONAL PARAMETERS

For all evolutionary algorithms, the operational parameters control the balance between exploitation (using the existing material in the population to best effect) and exploration (searching for better genes). These operators frequently interact with each other (Goldberg, 1989), and the optimal combinations are problem-dependent, and can be difficult to find. Fortunately, evolutionary algorithms have proven to be quite robust across wide ranges of these (Mayer, 2002). The key operators and parameters, and their applications in DE, are as follows.

### 3.1 Population Size

Price and Storn (1997) recommended a population size of 5 to 10 times the dimensionality of the problem, and stated 4 as a minimum value. In simulating crystal structures via DE, Weber and Bürgi (2002) used a population of 40 for a 7-dimensional problem (a factor of 5.7). These values have certainly been shown to work well in practice (Storn and Price, 1997), demonstrating that sufficient genetic material is contained in the populations. However, this could be using an excessive amount – research with other evolutionary algorithms (also using real-value coding) has produced best results with factors between 1.5 and 2 (Mayer, 2002). Values in this range could be more efficient, by carrying only a sufficient number of population members.

### 3.2 Selection of Parents

A large range of selection methods have been used in the past, including Roulette-wheel (the traditional choice for genetic algorithms), ranked, scaled, Queen-bee, complete, truncation, and tournament (where a size of two appears to be the current standard). Fortunately, in practice all forms appear to work well (Mayer, 2002). DE uses complete selection (each parent is considered in each generation), and this should perform adequately.

### 3.3 Recombination

A number of studies on evolutionary algorithms have shown recombination and mutation to have a synergistic effect (Michalewicz and Fogel, 2000). DE incorporates both of these into one operation using a form of uniform crossover, albeit in a somewhat more convoluted way than is normally used. Recombination is controlled in DE by the user-specified crossover rate (CR). For each parameter in turn, either the parent's allele is used, or a mutated allele (rather than a second parent's, as is usually the case in evolutionary algorithms). Storn and Price (1997) list CR values of 0.1 for a thorough (but slower) optimisation, to 1.0 for speedier (but risky) convergence, with 0.5 being recommended. Previous evolutionary algorithm studies have shown that most forms of recombination work well, across quite a wide range of rates, so 0.5 would appear an adequate first choice.

### 3.4 Mutation

DE has no defined mutation rate, instead taking this parameter as the flip-side of CR. Previous studies have shown low (around 0.01) to high (towards 1.0) rates to all be effective (Mayer, 2002). Using a CR of 0.5 gives a mutation rate of 0.5 also. Studies have shown that the exact form of mutation applied is less critical than ensuring that some form is present, to drive the exploration.

DE's unique form (which allows self-adaptation of the mutation sizes as the optimisation progresses) adds a scaled difference between two random parents to a third parent. This is an arithmetical form applied to each real-value 'gene' which may be intermediate or extrapolative, depending on the scaling factor (F). Storn and Price (1997) recommended an F of between 0.4 and 1, with 0.5 as a good initial choice. Investigations with DE (Kinghorn, unpublished) have found that 'pulsing' F to a larger amount, for example to 5.0, every few generations has the effect of assisting the optimisation process, as it induces extrapolative mutation.

### 3.5 Replacement Strategy

DE uses generational replacement, with elitism guaranteed in that the parents (in turn) are only replaced if their direct competitor is superior (or equal – this allows more genetic diversity to enter the search). This operation may not be as efficient as continuous deterministic replacement (as used in other evolutionary algorithms), but should suffice.

## 4. BEEF MODEL STUDY

The system simulated is a beef property in the northern speargrass region of Queensland, tuned to 'average' data from the Australian Bureau of Statistics. A stochastic individual-animal model (based on DYNAMA, a commercial herd management package, as described in Holmes, 1995) was used, with a daily time-step over a ten-year horizon. Further details of this model and parameters can be found in Mayer et al. (2001). The objective function is the ten-year accumulated gross margin (sales less variable costs), and there are 70 management options covering stocking rates, mating and weaning policies, and purchasing and culling decisions. This 70-dimensional problem has a search-space of the order of $10^{120}$.

Optimisations were run on a network of Sun workstations under Unix. Each month of runtime generates about $10^5$ model runs, with each run producing one value of the objective function. The longest optimisation so far totalled 150,000 runs, well short of the $10^6$ to $10^7$ required to give a high probability of finding the global optimum (Mayer et al., 2001). The latter stages are approaching convergence. In practical terms, these numbers are all that is currently computationally feasible, so any optimisation algorithms which perform well here can be recommended.

Genial has been the preferred evolutionary algorithm for conducting investigative optimisations of this system. Genial is a real-value genetic algorithm or evolution strategy (we have generally found the former implementation to be superior) with a wide range of operational parameters covering parent selection, replacement strategy, recombination and mutation. The most efficient of these optimisations used a comparatively small population size (200), and for this we had four replicates, using a range of recombination and mutation rates and types. The performance of these (Figure 1) shows little practical difference between these replicates. They all appear to be approaching convergence (but, disturbingly, towards different optima) at about $10^4$ model runs, with only a few 'minor' lifts occurring after this.

Also marked on Figure 1 is one DE optimisation. This used a population size of 250 (approximately the same as for the Genial replicates). This is only 3.5 times the dimensionality of the problem, whereas Storn and Price (1997) recommend a multiplier of 4 as a minimum. This optimisation had a crossover rate (and thus also mutation rate) of 0.5. The mutation factor was mostly 0.5, with an increase to 5.0 being applied every tenth generation.

In the initial stages of these searches (up to and past $10^4$ runs), the Genial results have a clear advantage over DE. However, DE keeps finding improved solutions in the mid and latter stages. There is one particularly notable 'lift' where the DE optimisation obviously found a more profitable region of the search-space to exploit. Here, DE surpasses the Genial optimisations, which are showing signs of sub-optimal convergence (these were apparently converging to different solutions, which is of obvious concern). We attribute at least some of DE's improvements to the self-adapting form of mutation used (versus the fixed types and rates as used in Genial), and particularly to the periodic application of extrapolative mutation to escape from sub-optimal regions.

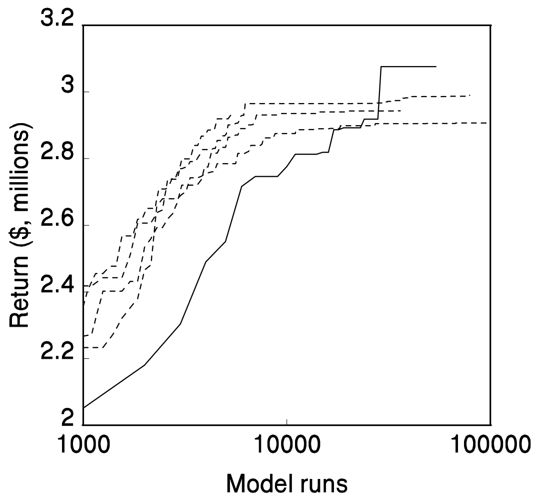

Figure 1. Performance of four replicates of Genial (dashed lines) using a population size of 200, and one of DE (solid line) with a population of 250.

Given the trend of higher efficiency with smaller population sizes, we then trialed DE with a population of 100 (only 1.4 times the dimensionality of this problem; well below that recommended by DE's developers). Two replicates were run – the first with extrapolative mutation (as previously used), and the second with this feature removed (using 'standard' interpolative mutation, but still with the self-adapting feature).

Figure 2 shows that these optimisations were quite successful – matching the Genial results in the initial stages, and then surpassing both these and the DE optimisation with a higher population size (250). The best combination for DE thus far is a population of 100 with extrapolative mutation, however the replicate with standard mutation is still running. We intend to take this out to $2\times10^5$ model runs, to test its 'longer-term' performance.

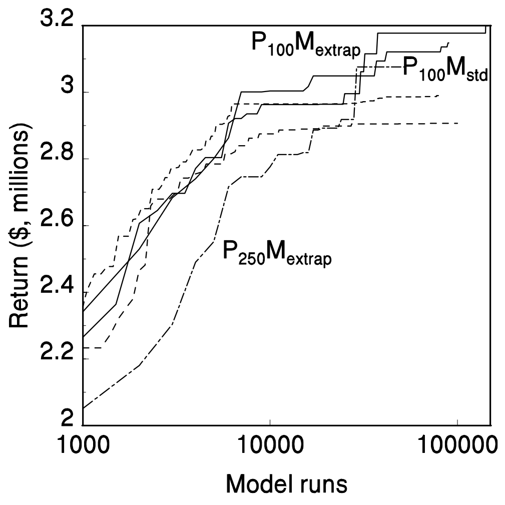

Figure 2. Two 'extreme' replicates of Genial (dashed lines) vs three DE optimisations (solid and dash-dot; parameters as indicated on graph – P for population size, M for mutation type).

## 5. CONCLUSIONS

DE is one of the more efficient evolutionary algorithms available. It requires few operational parameters, and these appear to be quite robust. DE is thus well suited to both novice and experienced users. In limited optimisations of both test functions and systems models, DE has performed as well as, or better than, a number of large and complex evolutionary algorithms that have been the recommended 'standards' up until now.

## 6. ACKNOWLEDGEMENTS

We are grateful to Meat and Livestock Australia for funding, and Katherine Delaney for the figures.

## 7. REFERENCES

Bratley, P., B.L. Fox and L.E. Schrage, *A guide to simulation*, Springer-Verlag, New York, 1987.

Coello Coello, C.A., D.A. Van Veldhuizen and G.B. Lamont, *Evolutionary algorithms for solving multi-objective problems*, Kluwer Academic Publishers, 515pp., Boston, 2002.

Goldberg, D.E., *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, Reading, 1989.

Glover, F., E. Taillard and D. de Werra, A user's guide to tabu search, *Annals of Operations Research* 41, 3-28, 1993.

Holmes, W.E., *BREEDCOW and DYNAMA – herd budgeting software package*, Queensland Department of Primary Industries, Townsville, 1995.

Kirkpatrick, S., C.D. Gelatt and M.P. Vecchi, Optimization by simulated annealing, *Science* 220, 671-680, 1983.

Mayer, D.G., *Evolutionary algorithms and agricultural systems*, Kluwer Academic Publishers, 107pp., Boston, 2002.

Mayer, D.G., J.A. Belward and K. Burrage, Robust parameter settings of evolutionary algorithms for the optimisation of agricultural systems models. *Agricultural Systems* 69, 199-213, 2001.

Meadows, D.H. and J.M. Robinson, *The electronic oracle*, Wiley, New York, 1985.

Michalewicz, Z. and D.B. Fogel, *How to solve it: Modern heuristics*, Springer, Berlin, 2000.

Nelder, J.A. and R. Mead, A simplex method for function minimisation, *The Computer Journal* 7, 308-313, 1965.

Price, K. and R. Storn, Differential evolution, *Dr. Dobb's Journal*, April, 18-24&78, 1997.

Storn, R. and K. Price, Differential evolution – A simple and efficient heuristic for global optimization over continuous spaces, *Journal of Global Optimization*, 11, 341-359, 1997.

Weber, T. and H.-B. Bürgi, Determination and refinement of disordered crystal structures using evolutionary algorithms in combination with Monte Carlo methods, *Acta Crystallographica A*, **58**, 526-540, 2002.

**Appendix 1. Pseudo-FORTRAN Coding of Differential Evolution.**

! Declare and initialise – INTEGERS are popsize {number of members in the population}, loci {number of alleles that each member contains}, generation {current}, max_gens {maximum number of generations}, i, j, k, a, b, c {counters}. REALS are score {value returned from model; to be optimised}, CR {crossover rate}, F {mutation factor}, value(popsize) {score values for the current parents}, parent_allele(popsize,loci) {alleles of the parents, initially generated at random and including a subroutine call to get scores for each}, progeny_allele(popsize,loci) {alleles of the progeny}, allele(loci) {temporary values, as created in DE}

```
do generation = 1, max_gens                        ! Loop for target number of generations

    do i = 1, popsize                              ! Loop for each population member

    1   call RANDOM_NUMBER(rand_num)
        a = INT(rand_num*popsize) + 1              ! Parent to be challenger's template
        if ( a.eq.i ) go to 1
    2   call RANDOM_NUMBER(rand_num)
        b = INT(rand_num*popsize) + 1              ! Choose two more parents
        if ( b.eq.i .or. b.eq.a ) go to 2
    3   call RANDOM_NUMBER(rand_num)
        c = INT(rand_num*popsize) + 1              ! Parents used to construct challenger
        if ( c.eq.i .or. c.eq.a .or. c.eq.b ) go to 3      ! must all be different

        call RANDOM_NUMBER(rand_num)
        j = INT(rand_num*loci) + 1                 ! Random start for loci cycle
        do k = 1, loci                             ! Loop for each loci
            call RANDOM_NUMBER(rand_num)
            if ( rand_num.lt.CR .or. k.eq.loci ) then      ! MUTATION
                allele(j) = parent_allele(c,j) + F * (parent_allele(a,j) – parent_allele(b,j))
            else
                allele(j) = parent_allele(i,j)     ! CROSSOVER
            end if
            j = j + 1
            if ( j.gt.loci ) j = j - loci
        end do                                     ! End loop for each loci

        call Ag_Model(loci, allele, score)         ! Evaluate – run model with allele as inputs
        if ( score.ge.value(i) ) then
            value(i) = score                       ! If competitor is better, replace parent
            do j = 1, loci
                progeny_allele(i,j) = allele(j)
            end do
        else                                       ! or
            do j = 1, loci
                progeny_allele(i,j) = parent_allele(i,j)   ! Current parent carries thru to next generation
            end do
        end if

    end do                                         ! End loop for each population member

    score = -9E15
    do i = 1, popsize                              ! Loop for the new population
        if ( value(i).gt.score ) then
            k = i                                  ! k is index number of the best member
            score = value(k)
        end if
        do j = 1, loci
            parent_allele(i,j) = progeny_allele(i,j)       ! Progeny become new parents
        end do
    end do                                         ! End loop for new population
    write value(k), (allele(k, i), i=1, loci)      ! Report best solution (or every few gens.)

end do                                             ! End loop for target number of generations
```