Evolutionary Computing for Operating Point Analysis of Nonlinear Circuits

Mark Zwolinski[#], Duncan Crutchley[#], Zheng Rong Yang[+]

[#] Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK, email: mz@ecs.soton.ac.uk

[+] Department of Physics, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, Scotland.

ABSTRACT

The DC operating point of an electronic circuit is conventionally found using the Newton-Raphson method. This method is not globally convergent and can only find one solution of the circuit at a time. In this paper, evolutionary computing methods, including Genetic Algorithms, Evolutionary Programming, Evolutionary Strategies and Differential Evolution are explored as possible alternatives to Newton-Raphson. These techniques have been implemented in a trial simulator. Results are presented showing that Evolutionary Computing methods are globally convergent and can find multiple solutions to circuits. The CPU time for these new methods is poor compared with Newton-Raphson, but better implementations and the use of hybrid methods suggest that further work in this area would prove fruitful.

Keywords: Circuit simulation, Genetic Algorithms, Evolutionary Computing, Differential Evolution

1. INTRODUCTION

The first task in simulating the behaviour of a circuit is to find the DC operating point. Traditionally, this has been done using the *Newton-Raphson* (NR) method. NR has three potential problems. First, at the start of each iteration we must re-compute the Jacobian matrix (of partial derivatives), which is computationally costly. Second, the solution can diverge or even oscillate. Moreover, convergence is only guaranteed if a suitable initial solution vector is chosen to begin the simulation. For circuits with more than one possible solution, the initial guess can influence the final solution. Finally, finding multiple global solutions is generally impossible.

In this paper we discuss various aspects of *Evolutionary Computing* (EC), in particular, *Evolution Strategies* (ES) and *Genetic Algorithms* (GA), which will be seen to have

The work reported in this paper has been supported by EPRSC.

advantages over NR for DC circuit analysis. The main benefit is improved convergence and the ability to find multiple solutions of appropriate nonlinear circuits. This can be attributed to the parallel nature of EC algorithms i.e. searching through a population of solutions rather than a sequential search for individual solutions, as in NR.

We represent the trial solutions (the node voltages) as a real-valued *trial vector* $\mathbf{x}^k = (x_0^k,...,x_{n-1}^k)^T$, at iteration *k*. The aim is to find a set of *n* variables $\mathbf{x}^* = (x_0{}^*,...,x_{n-1}{}^*)^T$, such that, for some *objective function f*, we have $f(\mathbf{x}^*) = optimum$, where $f(\mathbf{x}^k) \in \Re$ and $\mathbf{x}^k \in \Re^n$. Traditional methods such as NR define the objective function as a vector $\mathbf{f} \in \Re^n$ based on the equations of the nonlinear circuit components. In the case of EC algorithms the objective function *f* represents the *fitness* of a particular trial vector.

In NR we solve the set of nonlinear equations $\mathbf{f}(\mathbf{x}^k) = \mathbf{0}$, iteratively as $\mathbf{x}^{k+1} = \mathbf{x}^k - \nabla_k^{-1} \cdot \mathbf{f}(\mathbf{x}^k)$, where $\nabla_k = \left[\frac{\partial f_i}{\partial x_j^k}\right]_{i,j}, i, j = 0,1,...,n-1$ is the Jacobian matrix at iteration *k*.

Several techniques have been proposed to aid the convergence of NR. One particular difficulty with NR is its sensitivity to the initial settings of the solution vector. This is especially a problem when we are dealing with nonlinear circuit equations that often have multiple solutions. In this case different initial settings can result in convergence to a different solution or even to divergence. There are several techniques that can be used to help convergence, such as damping algorithms [1], source stepping algorithm [2], the $G_{\min}$ stepping procedure [3] and homotopy [4].

2. EVOLUTIONARY COMPUTING

In general, when using EC algorithms, for each member of the *population* we aim to optimise a set of *p* objectives with or without up to *q* additional constraints that need not be optimised but neither shall be degraded. For the purpose of DC analysis $p = n$ (the number of node voltages) and $q = 0$. We denote the objectives by the *objective vector*, $\mathbf{y}^{i,k} = \left(y_0{}^{i,k}, y_1{}^{i,k},..., y_{p-1}{}^{i,k}\right)^T$. Here, $k = 0,1,...,\text{KMAX}-1$ denotes the generation count, and $i = 0,1,...,NP-1$ is the position in the population. *NP* is the population size, equal here to *n*. In the case of DC analysis we define y_m as the net current flowing into node *m*. During the optimisation process we aim to minimise the y_m parameters, hence as a trial vector reaches optimality, the y_m values in $\mathbf{y}^{i,k}$ tend to zero. As we have not defined any constraints, $\mathbf{y}^{i,k}$ is an *n*-dimensional vector, its components are in one-to-one correspondence with those of $\mathbf{x}^{i,k}$, the i^{th} trial vector of node voltages at the k^{th} generation.

2.1 Fitness Functions

EC algorithms are designed to work on a population of trial vectors and exhibit implicit parallelism. To enable the processes of evolution to be simulated, it is necessary for each member of the population to be assigned a value representing the worth of the solution. This is called the fitness of the individual or trial vector; the fitness can then be used to

decide which trial vectors in the population survive from one generation to the next. There are many choices for the fitness function; four are described here. In practice, some work better than others. The general technique is to use $\mathbf{x}^{i,k}$ and other knowledge of the circuit to compute $\mathbf{y}^{i,k}$. We use $\mathbf{y}^{i,k}$, which is the error vector for $\mathbf{x}^{i,k}$, to obtain a fitness score. The overall optimisation procedure then aims to minimise the fitness scores.

The first fitness function (FF1) uses the root mean square value of the components of $\mathbf{y}^{i,k}$. In other words, we have the following function.

$$f_{\mathrm{FF1}}(\mathbf{x}^{i,k}) = \mathrm{RMS}(\mathbf{y}^{i,k}) = \sqrt{\frac{\sum_{m=0}^{p-1} \left(y_m^{i,k}\right)^2}{p}} \tag{1}$$

The second fitness function (FF2) is similar to FF1 and is the sphere function.

$$f_{\mathrm{FF2}}(\mathbf{x}^{i,k}) = \mathrm{SPHERE}(\mathbf{y}^{i,k}) = \sum_{m=0}^{p-1} \left(y_m^{i,k}\right)^2 \tag{2}$$

This is sometimes described as the sum of the squared errors and is a common choice for fitness function. It is important to be aware that FF2 has the ability to hide the path to the global optima in certain applications [5] and this is obviously an undesirable effect in global optimisation problems such as circuit simulation.

FF3 and FF4 are, in practice, better choices for the fitness function. Both of these involve a weighting scheme to give bias to the important components of $\mathbf{y}^{i,k}$. FF3 is a weighted sum defined as follows.

$$f_{\mathrm{FF3}}(\mathbf{x}^{i,k}) = \mathrm{wSUM}(\mathbf{y}^{i,k}) = \sum_{m=0}^{p-1} w_m \cdot y_m{}^{i,k} \tag{3}$$

Here w_m is a real-valued weight factor attached to the m^{th} component of $\mathbf{y}^{i,k}$. FF3 is valid if and only if the solution space is convex [6].

FF4 also uses a weighting factor, this time on the maximum error score.

$$f_{\mathrm{FF4}}(\mathbf{x}^{i,k}) = \mathrm{wMAX}(\mathbf{y}^{i,k}) = \max(w_m \cdot y_m{}^{i,k}) \tag{4}$$

FF4 usually provides the best choice of fitness function due to its min-max formulation, which guarantees that all local minima and, in the majority of cases, the global minimum can be found [6].

2.2 Genetic Algorithms

Genetic algorithms are randomly guided probabilistic heuristic search algorithms based on the mechanics of natural selection and natural genetics. The relation between the biological terms and GAs are shown in Table 1.

When using GAs for DC analysis the phenotypes are equivalent to the real-valued trial vectors $\mathbf{x}^{i,k}$. We use $\mathbf{g}^{i,k}$ to denote the genotype corresponding to $\mathbf{x}^{i,k}$. The genotypes take the form of a two-part array $\mathbf{g}^{i,k} = [\mathbf{g}_I{}^{i,k}, \mathbf{g}_F{}^{i,k}]$, where $\mathbf{g}_I{}^{i,k}$ contains n binary strings

(chromosomes) representing the integer parts of $\mathbf{x}^{i,k}$ and $\mathbf{g}_F{}^{i,k}$ contains *n* binary strings representing the fractional parts of $\mathbf{x}^{i,k}$. The genes are the single bits that form these chromosomes. The allele values are 0 and 1. One can map the genotypes onto the corresponding phenotypes using a suitably defined mapping function $\xi(\mathbf{g}^{i,k}) = \mathbf{x}^{i,k}$.

Table 1. Biological terms used in genetics and their GA counterparts.

Biological Term	GA Analogue
chromosome	bit string – containing genes
gene	feature of a chromosome
allele	feature value e.g. 0 or 1
genotype	structure of one or more chromosomes
phenotype	decoded structure – the real valued quantifier of the corresponding genotype

Basic GAs use three operators to create new offspring. They are *crossover*, *mutation* and *inversion*. Crossover is used to perform sexual reproduction. In its simplest form it requires choosing two parents at random and a cutting point, ζ, (a position along the chromosome) the resulting partial strings from the parents are cross-spliced to form two new offspring.

Crossover is similar in principle to the recombination operator in evolutionary strategies (section 2.3) and allows genetic material from the two parents to be passed onto their offspring. This enables the propagation of strong characteristics of the parents to survive through to the next generation in the form of even fitter offspring. Furthermore, this means that if the two parents are sub-optimal then it is still possible to create strong offspring by using crossover.

Mutation is an operator that acts on a single parent. In its simplest form a gene (a single bit) is randomly chosen from the parent's chromosome(s) and negated. When used as a secondary operator mutation helps to explore areas of the solution space that may be missed by the large changes made by crossover. Historically, the mutation rate is set to $1/l$ (where l is the length of chromosome), which can be a very small number when l is large.

The final operator is *inversion*, which also acts on a single parent. It works as a reordering operator that aims to protect good genetic material that is widely spaced along chromosomes and that might be lost by later crossover [7]. The basic principle is to pick two random cutting points in a chromosome. The partial string contained between these points is reordered (usually it is reversed) and the resultant chromosome becomes the offspring.

Historically, it has been thought that the primary operator in GAs should be crossover [7] because of its effective use in the natural world. More recently it has been found that it is often advantageous to use crossover as a secondary operator and instead use mutation as the primary operator [8]. The benefits of using inversion are unclear and if used should be a tertiary operator with respect to crossover and mutation.

The GA starts by generating an initial population consisting of two pools: one containing μ parents and the other containing places for λ offspring (initialised in the same way as the parent pool). We only need to generate the genotypes because we can use the mapping function ξ to obtain the phenotypes. An operator and then a parent or parents are

randomly chosen. The offspring are placed in the offspring pool and once the pool is full the whole population is reordered – fittest first – such that the first μ trial vectors form the new parent pool. This process continues until the solutions have reached the required accuracy.

A frequent problem arising with GAs is premature convergence. This happens when the chromosomes contained within the population reach a point where crossover no longer produces offspring that can out-compete their parents, which is necessary for a homogeneous population. If this happens then the crossover operators will only succeed in regenerating the current set of parents! Further optimisation then has to rely solely on the mutation operator, which can of course be slow. One other frequent failing of GAs is *stagnation* or the *trap phenomenon* where the algorithm stagnates at a point that may or may not be close to an optimal solution.

The initial settings for GA are problem-dependent and in the case of DC analysis can vary from one circuit to another. Typically, the crossover probability is in the range 0.08 to 0.25 and the mutation probablity in the range 0.5 to 0.9. When generating the genotypes we use the range $[-32767.0, 32767.0]$ for the components of $\mathbf{g}_I^{i,k}$ and for the components of $\mathbf{g}_F^{i,k}$ we use the range $[0, 1-2^{-31}]$.

2.3 Evolutionary Programming and Evolution Strategies

Both *Evolutionary Programming* (EP) [9] and *Evolution Strategies* (ES) [10] are probabilistic heuristic direct search optimisation techniques like GAs. EP and ES do not work at the genetic level, but instead operate at the phenotypic level, which has distinct advantages for real-valued problems because there is no longer a need to define genotype representations and genotype-to-phenotype mapping functions. There is, in general, no crossover or inversion in ES or EP. Sometimes it can be beneficial to have some crossover-like operation, when this is the case we use recombination. The cutting points for recombination are simpler than those for GAs.

Evolutionary Programming was proposed in 1962 by Fogel [9]. A population of finite-state machines is used to predict input symbols. As each input symbol is offered to each parent machine, each output symbol is compared to the next input symbol. The fitness of the symbol prediction is then evaluated. The average fitness per symbol represents the fitness of the state machine. The fittest parent machines are allowed to produce one offspring by mutation.

The method of Evolution Strategies is the real-valued counterpart to EP. The ES method uses a population divided into two pools, where the first μ members of the population form the parent pool and the remaining λ members form the offspring pool. The parent pool consists of trial vectors $\mathbf{x}^{i,k}$ of variables uniformly distributed over the possible solution range.The offspring pool is initialised in the same way. In each generation, parents are randomly selected to create offspring by a single reproduction operator, usually mutation. Parents are chosen at random from the parent pool to generate one offspring, with the possibility that a parent can be chosen again later in the same generation to create further offspring. We then reorder the entire population in order of fitness. The μ fittest will now form the parent pool. We denote this as $(\mu + \lambda)$ - ES. Alternatively, we can update the parent

pool with the μ best trial vectors of the λ offspring and discard the previous generation; we denote this as (μ, λ) - ES . This latter scheme is not used here.

ES mutates a parent by adding a Gaussian-distributed random vector with mean zero and predefined standard deviation [8].

$$\tilde{\mathbf{x}}^{i,k} = \mathbf{x}^{i,k} + \mathbf{u}^{i} \tag{5}$$

Here the *mutation vector* $\mathbf{u}^{i}$ is computed as:

$$\begin{aligned} \mathbf{u}^{i} &= (u_0^i, u_1^i, ..., u_{n-1}^i)^T \\ u_j^i &= N_j(0, \sigma) \end{aligned} \tag{6}$$

In equation (6) $\sigma = \tau.\sigma_k$ represents a predefined deviation or step size of the mutation vector at generation *k* and τ is a user-set scale factor. σ_k is the standard deviation of the population at generation *k*.

The basic ES method uses the same standard deviation to generate each variable in all the mutation vectors in a single generation. This is not very realistic, it is perhaps better to have a different step size for each of the variables. This allows for more diverse solutions and a better exploration of the solution space [8]. If one implemented this directly many user-set parameters are needed (one for each variable in $\mathbf{u}^{i}$), hence it is useful if the step sizes can self-adapt, thus letting the algorithm find the best settings [11].

One possible self-adaptive technique for mutation, used here, is:

$$\begin{aligned} u_j^i &= N(0, \sigma_{k+1}^j) \\ \sigma_{k+1}^j &= \sigma_k^j \cdot \exp\left(\tau' \cdot N(0,1) + \tau \cdot N_j(0,1)\right) \end{aligned} . \tag{7}$$

This provides a different standard deviation for each variable in $\mathbf{u}^{i}$. Overall, we form multiple *deviation vectors* $\boldsymbol{\sigma}_{i,k}$, from the σ_k^j generated by (7). Thus we have one $\boldsymbol{\sigma}_{i,k}$ for each trial vector $\mathbf{x}^{ik}$. The variable $N(0,1)$ is a standard Gaussian random deviate globally set and regenerated at the start of each generation and $N_j(0,1)$ is the j^{th} independent identically distributed standard Gaussian random deviate. The parameters τ and τ' are defined as [11]:

$$\tau = \zeta / \sqrt{2n}, \tau' = \zeta / \sqrt{2\sqrt{n}}, \tag{8}$$

ζ is a user set scale factor. Other, similar, self-adaptive mutations are possible [8].

These two approaches to ES only use mutation. As discussed in section 2.2, crossover is often very useful because it helps to propagate strong genetic material from parents to offspring. In ES this is called recombination. In this work, the basic ES does not use any recombination, but the self-adaptive ES algorithm has been designed to use one of four recombination operators: RECOM0 – no recombination, RECOM1 – average recombination, RECOM2 – intermediate recombination or RECOM3 – discrete recombination. These are defined as follows.

RECOM1: Two parents $\mathbf{x}^{i,k}$ and $\mathbf{x}^{j,k}$, $i \neq j \in 0,1,...,\mu-1$ are randomly selected before mutation. We compute the average of the two vectors, $\mathbf{v}$, and the average of the corresponding deviation vectors $\boldsymbol{\sigma}_{i,k}$ and $\boldsymbol{\sigma}_{j,k}$, $\overline{\boldsymbol{\sigma}}$.

RECOM2: Two parents $\mathbf{x}^{i,k}$ and $\mathbf{x}^{j,k}$, $i \neq j$ are randomly selected before mutation the recombined intermediate vector $\mathbf{v}$ and the intermediate deviation vector $\overline{\boldsymbol{\sigma}}$ are computed as:

$$\begin{aligned} \mathbf{v} &= \mathbf{x}^{i,k} + r \cdot \left(\mathbf{x}^{j,k} - \mathbf{x}^{i,k}\right) \\ \overline{\boldsymbol{\sigma}} &= \boldsymbol{\sigma}_{i,k} + r.\left(\boldsymbol{\sigma}_{j,k} - \boldsymbol{\sigma}_{i,k}\right) \end{aligned} \tag{9}$$

In equation (9), variable r is a uniformly distributed random deviate between 0 and 1.0.

RECOM3: Two parents $\mathbf{x}^{i,k}$ and $\mathbf{x}^{j,k}$, $i \neq j$ are randomly selected before mutation. One then forms the intermediate vector $\mathbf{v}$ by randomly picking components from the parent vectors, i.e. randomly choosing the m^{th} component from $\mathbf{x}^{i,k}$, or from $\mathbf{x}^{j,k}$, where $m = 0,1,...n-1$. We do the same with the deviation vectors of both parents.

After recombination we place $\mathbf{v}$ in the offspring pool and refill the parent pool by taking the fittest overall individuals from the offspring and the current parent pools. Typically, $\mu = 200$, $\lambda = 200$ and $\tau = 0.5$ with a generation limit of 10000.

2.4 Differential Evolution

Self-adaptation adds to the robustness of evolutionary algorithms by reducing user interaction. Storn and Price developed an evolutionary algorithm called *Differential Evolution* (DE) [7] that is self-adaptive, simple and yet very powerful. The method is perhaps the simplest evolutionary algorithm to implement and has been shown to be one of the most robust methods [5]. The evolutionary methods described are not guaranteed to converge to the global optimum. They can stagnate at local optima. This can be avoided by using self-adaptation and variable mutation rates. DE uses multiple trial vectors and the differences between these vectors are used to set parameters such as step size. Several DE schemes have been proposed by Storn [12], but here, only two schemes will be discussed.

In DE1 [6], for each trial vector $\mathbf{x}^{i,k}$**,** we generate an intermediate vector $\mathbf{v}^i$ as:

$$\mathbf{v}^i = \mathbf{x}^{r_1,k} + \tau \cdot (\mathbf{x}^{r_2,k} - \mathbf{x}^{r_3,k}) \tag{9}$$

where τ is a positive real-valued, user-set scale factor and r_1, r_2 and r_3 are randomly selected, mutually distinct integers in the range $[0, NP-1]$. The intermediate vector $\mathbf{v}^i$ is then used with $\mathbf{x}^{i,k}$ to generate a new offspring $\tilde{\mathbf{x}}^{i,k}$. If $\tilde{\mathbf{x}}^{i,k}$ is fitter than $\mathbf{x}^{i,k}$ then $\mathbf{x}^{i,k+1} = \tilde{\mathbf{x}}^{i,k}$ and we discard $\mathbf{x}^{i,k}$, else we keep $\mathbf{x}^{i,k}$. We generate offspring using the following formula.

$$\tilde{\mathbf{x}}^{i,k} = \begin{cases} v_j^i, \text{ for } j = \langle K \rangle_n, \langle K+1 \rangle_n, ..., \langle K+L-1 \rangle_n \\ x_j^{i,k}, \text{ otherwise} \end{cases} \tag{10}$$

In equation (10), K is a randomly selected integer in the range $[0, n-1]$. L is an integer in the same range but with the probability $\Pr(L = r) = c^r$, where c is the user-set crossover probability, $c \in [0.0, 1.0]$. $\langle K \rangle_n$ denotes $K \bmod n$.

DE2 is identical to DE1 except for the generation of the intermediate vector, $\mathbf{v}^i$. An additional difference vector is used.

$$\mathbf{v}^i = \mathbf{x}^{i,k} + \tau' \cdot (\mathbf{x}^{\text{best},k} - \mathbf{x}^{i,k}) + \tau \cdot (\mathbf{x}^{r_1,k} - \mathbf{x}^{r_2,k}) \tag{11}$$

This time we only need two random integers r_1 and r_2, and τ' is another positive user-set scale factor. By including the extra difference vector, involving the current generation's best solution, we enhance the greediness of the algorithm. This scheme also has benefits when used with objective functions such as FF4 that are not constructed from many parameters. DE1 is usually best for general use and works well for FF1, FF2, FF3 and FF4.

When using DE there are several rules that, where possible, should be obeyed to improve performance. For instance, it is suggested [12] that the initial population should be spread over the full range of the problem variables, e.g. $[-V_{DD}, V_{DD}]$ in the case of DC analysis. Usually c should be set to a value less than 0.5 but if the algorithm fails to converge then c can be increased to 1.0. As an initial guess, the best population size is usually $\text{NP} = 10n$ and the user should try $\tau, \tau' \in [0.5, 1.0]$. As NP is increased above $10n$, τ and τ' should be decreased. The best choice of fitness function is generally FF4 but this can yield a lot of local optima. Typical values for DE1 are $c = 0.5$ and $\tau = 0.7$ and for DE2 $c = 0.3$, $\tau = 0.85$ and $\tau' = 0.95$ with a generation limit, as before of 10000.

3. RESULTS

In order to test the various algorithms, a basic circuit simulator has been written in ANSI C with interchangeable front ends; one for each of the techniques: Newton-Raphson, Genetic Algorithm (fixed and variable mutation rates), Evolutionary Strategy, Self-Adaptive Evolutionary Strategy (with four possible types of recombination) and Differential Evolution (DE1 and DE2). Four CMOS test circuits were used to evaluate the performance of each of these techniques. The circuits are: an RS-Latch with set and reset at logic 1 e.g. S=R=1; a simple D-latch with clock C=1 and D=0; a Transmission Gate XOR with inputs A=0 and B=0 and a Positive Edge-Triggered D Flip-Flop with S=R=D=C=1. Inputs A=1 B=0 were also tried for the XOR but NR failed – all of the EC methods did find a solution but due to the failure of NR an accurate error assessment could not be performed. Hence the table for this configuration of the XOR is omitted. The RS-latch contains 8 transistors, the D-latch contains 18 transistors, the XOR has 6 transistors and the Flip-Flop comprises 36 transistors. The six algorithms use parameter settings suggested above, e.g. population size, mutation rates etc.

The results are presented in four separate tables, one for each circuit, to compare the performance of the various algorithms. The columns of the table labelled *Error* give the error, in Volts, of the voltages at nodes in the circuit for which a DC operating point is required. The average error per node, along with the maximum and minimum errors are given. The errors are computed with respect to the NR solutions. Some of the test circuits have multiple solutions such as the RS-latch, D-latch and Flip-Flop. When an algorithm has found multiple solutions then this has been noted in the table. The XOR only has one solution for a given set of inputs.

Table 2. Results For RS-Latch (R=S=1)

Met-hod	Best FF	Best REC (ESA)	Mutn. Rate (GA)	No. Of Solns	No. Of Gens./ Iterns.	Av.	Error Min.	Max	CPU Time (s)
NR	~	~	~	1	26	~	~	~	0.0004
DE1	FF4	~	~	2	2735	0.1876	0.2657	0.0008	0.126
DE2	FF4	~	~	2	334	0.0077	0.0369	0.0005	0.46
ES	FF4	~	~	2	13	0.0015	0.1354	0.0299	19.3
ESA	FF2	0	~	2	36	0.0258	0.0347	0.0034	0.33
GA	FF1	~	FIXED	1	876	0.0360	0.1245	0.0043	13.3

Table 3. Results For Simple D-latch (C=1 D=0)

Met-hod	Best FF	Best REC (ESA)	Mutn. Rate (GA o)	No. Of Solns	No. Of Gens./ Iterns.	Av.	Error Min.	Max	CPU Time (s)
NR	~	~	~	1	8	~	~	~	0.0003
DE1	FF4	~	~	2	3784	0.0602	0.4080	0.0001	0.69
DE2	FF3	~	~	2	2788	0.0715	0.4405	0.0000	13.7
ES	FF4	~	~	1	31	0.1044	0.6086	0.0033	1.5
ESA	FF3	1	~	1	350	0.0635	0.4568	0.0001	6.6
GA	FF2	~	FIXED	1	1656	0.1245	0.6474	0.0006	40.4

Table 4. Results For XOR (A=B=0)

Met-hod	Best FF	Best REC (ESA)	Mutn. Rate (GA)	No. Of Solns	No. Of Gens./ Iterns.	Av.	Error Min.	Max	CPU Time (s)
NR	~	~	~	1	7	~	~	~	0.00009
DE1	FF3	~	~	1	69	0.0213	0.0576	0.0028	0.002
DE2	FF4	~	~	1	70	0.0123	0.0302	0.0018	0.033
ES	FF4	~	~	1	7	0.2515	0.5622	0.0002	0.058
ESA	FF3	1	~	1	45	0.0070	0.0190	0.0001	0.35
GA	FF2	~	FIXED	1	34	0.0270	0.1198	0.00008	0.47

Table 5. Results For D Flip-Flop (C=D=R=S=1)

Met-hod	Best FF	Best REC (ESA)	Mutn. Rate (GA)	No. Of Solns	No. Of Gens./ Iterns.	Av.	Error Min.	Max	CPU Time (s)
NR	~	~	~	1	33	~	~	~	0.005
DE1	FF3	~	~	1	5895	0.6232	2.1552	0.0005	45.5
DE2	FF4	~	~	1	2996	0.3335	1.3001	0.0188	54.5
ES	FF3	~	~	1	62	0.2580	0.6234	0.0017	6.8
ESA	FF4	1	~	1	310	0.3107	0.8179	00138	13.5
GA	FF2	~	FIXED	1	3592	0.6812	1.8312	0.0478	127.8

The algorithms were tested using all the possible configurations. For example, ESA was tested using all the fitness functions and for each fitness function all the possible recombination operators were tried. Different parameter settings were also tried in an attempt to obtain optimum performance. The same approach to testing was applied to each algorithm, for each test circuit and where possible the most reliable configuration of input parameters of an algorithm was used to obtain a fair comparison. As a result, a large amount of test data was generated and could not practically be included in this paper. Hence, the tables contain the data for the best run of each algorithm on each circuit. Parameter settings have been omitted from the tables because they are generally those suggested in earlier sections but can vary slightly from problem to problem.

From the above tables we can see that NR is by far the quickest of all of the methods and in the majority of cases GA performs worst in terms of both speed and accuracy. GA was the most sensitive to parameter settings. The poor performance of GA is expected mainly because of the need for special representations of the solution vector components. The best choice of fitness function was FF4 or FF3. Even when FF1 or FF2 worked best, FF3 and FF4 were not far behind.

From these results, DE1 and DE2 give the best results in terms of the number of solutions found. Furthermore, DE1 and DE2 yield accurate results in the majority of cases, although the self-adaptive ES also has good accuracy. The CPU time for the algorithms can vary dramatically from circuit to circuit, sometimes as a result of small changes to input parameters. Hence, when using these methods one should weigh up whether speed is the most important point or whether getting multiple solutions is more important. Overall, DE (1 or 2) seems to be the best general evolutionary algorithm because they are very simple to implement, they are both compact code-wise and they also have the least memory overhead because of the small population size.

4. CONCLUSIONS

The use of Evolutionary Computing algorithms for nonlinear operating point analysis of MOS circuits has been described. It has been demonstrated that EC and particularly Differential Evolution has some notable advantages over conventional NR. In principle, DE and the other EC algorithms are globally convergent, whereas NR is only locally convergent. It has been shown that DE can find multiple solutions in a single pass. It has also been seen that all of the EC techniques are sensitive, by varying degrees, to reproduction parameters, such as the mutation rate, population size, recombination strategies etc. The success of DE is partly due its self-adaptive nature and although DE uses mutation as a primary operator it also contains a recombination operator. Another excellent feature of the DE algorithms is that the population size is automatically scaled in proportion to the size of the given problem, which can help avoid over- and undersized populations. These features and the way they are implemented in DE have been the major contribution to DE's good performance.

All the EC techniques here are slow compared with NR even though the Jacobian matrix is not constructed. This can be attributed to the large populations that are required to run the algorithms. The accuracy is not as good as NR and in some cases the error can be quite significant.

Evolutionary techniques are normally globally convergent but the quality of the solutions can vary. NR can diverge and is only locally convergent but the solutions are extremely accurate. It is reasonable to ask whether a hybrid method is possible so that we have the best of both worlds. Such a method has been proposed by Salomon [13] called *Evolutionary-Gradient-Search* (EGS).

Future work will include devoting time to increasing the performance of the best algorithm, namely DE, in terms of convergence speed and accuracy. The next stage in development will require testing the DE algorithms on significantly larger circuits. This in turn will require a more sophisticated circuit simulator. The DE algorithm will, therefore, be integrated into such a simulator. This will be beneficial, firstly, because we will be able to build a much wider class of circuits and, secondly, we will see just how feasible EC is as a solution method for DC analysis of large circuits. The use of EC for other types of circuit simulation, such as *Transient Analysis*, will also need to be explored and can be included as part of the integration process into a SPICE-type simulator. Once this integration has been completed the main improvement to the EC method will be to reduce the CPU time, at present the EC algorithms are at best two orders of magnitude slower than NR.

REFERENCES

[1] Ho, C.W., Zien, D.A., Ruehli, A.E. and Brennan, P.A., An Algorithm for DC Solutions in an Experimental General Purpose Interactive Circuit Design Program, IEEE Trans. on Circuits and Simulation, Vol. CAS-24, No. 8, August 1977.

[2] Broyden, C.G., A Class of Methods for Solving Nonlinear Simultaneous Equations, Math Comp., Vol. 19, 1965, pp 577-593.

[3] Najibi, T.N., Continuation Methods as applied to Circuit Simulation, IEEE Press Circuits and Devices Magazine, Vol. 5, No.5, 1989, pp 48-49.

[4] Trajkovic, L., Homotopy methods for computing dc-operating points in Encyclopedia of Electrical and Electronics Engineering, vol. 9, pp. 171-176, 1999, John Wiley & Sons.

[5] Storn, R. and Price, K., Minimizing the Real Functions of the ICEC '96 Contest by Differential Evolution, *Proceedings Int. Conf. On Evolutionary Computing*, Nagoya. 1996

[6] Storn, R. and Price, K., Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces, Tech Report TR-95-012, ICSI, Berkeley, 1995

[7] Goldberg, D.E., Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley, 1989.

[8] Fogel, D.B., Evolutionary Computation: Towards a New Philosophy of Machine Intelligence, 2nd Ed., IEEE Press, NY, 2000.

[9] Fogel, L.J., Autonomous Automata, Industrial Research, Vol. 4, 1962, pp 14-19.

[10] Rechenberg, I., Cybernetic Solution Path of an Experimental Problem, Royal Aircraft Establishment, Library Translation No. 1122, August 1965.

[11] Bäck, T. and Schwefel, H.-P., An Overview of Evolutionary Algorithms for Parameter Optimization, Evolutionary Computation, Vol. 1:1, 1993, pp 1-23.

[12] Storn, R., On the Usage of Differential Evolution for Function Optimization, Technical Report, ICSI, Berkeley, 1996.

[13] Salomon, R., Evolutionary Algorithms and Gradient Search: Similarities and Differences, IEEE Trans. on Evolutionary Computation, Vol. 2, No. 2, July. 1998, pp 45-55.